VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Symbolická verifikace (SMV-tool)

# Symbolic verification (SMV-tool)

2009 Bc. Zdeněk Janoš

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2009 .....

Rád bych na tomto místě poděkoval Prof. RNDr. Petrovi Jančarovi, CSc., který mi s prací pomohl, protože bez něj by tato práce nevznikla.

## Abstrakt

Práce předkládá teoretické základy a informace k nástroji Symbolic Model Verifer (SMV). Tento nástroj je založen na binárních rozhodovacích diagramech. V rámci této práce jsou tedy objasněny základní pojmy týkající se binárních rozhodovacích diagramů (BDD), jejich modifikací, převodu na uspořádané binární rozhodovací diagramy (OBDD) a jejich redukcí na redukované uspořádané binární rozhodovací diagramy (ROBDD). S využitím poznatků týkajících se logiky, automatů, verifikace a binárních rozhodovacích diagramů je v této práci popsáno modelování systémů, které jsou následně verifikovány v nástroji SMV. SMV nástroj umožňuje symbolické ověřování modelů konkrétních systémů. Na názorných příkladech jsou v této práci ukázána možná využití nástroje SMV a jeho případná omezení.

**Klíčová slova:** konečné automaty, binární rozhodovací diagramy, uspořádané binární rozhodovací diagramy, redukované uspořádané binární rozhodovací diagramy, temporální logika, symbolické ověřování modelu, symbolická verifikace, SMV, nástroje pro symbolickou verifikaci

## Abstract

This thesis brings theoretical principles and information to the tool Symbolic Model Verifier (SMV). This tool is based on Binary Decision Diagrams. Within the scope of this thesis are clarified a basic notifications concerning Binary Decision Diagrams (BDD), their modifications and transmission to Ordered Binary Decision Diagrams (OBDD) and their reductions in Reduced Ordered Binary Decision Diagrams (ROBDD). With usage piece of knowledge concerning logic, automatics, verification and Binary Decision Diagrams is in this thesis described modelling systems, which are subsequently verification in the SMV tool. The SMV tool allows a symbolic model checking of concrete systems. On the objective examples are in this thesis shown possibility of usage of the SMV tool and its appropriate limitation.

**Keywords:** finite state machine, binary decision diagrams, ordered binary decision diagrams, reduced ordered binary decision diagram, temporal logic, symbolic model checking, symbolic verification, SMV, symbolic verification tools

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| BDD | – | Binary decision diagram |
| | – | Binární rozhodovací diagram |
| CNF | – | Conjunctive normal form |
| | – | Konjunktivní normální forma |
| CTL | – | Computation tree logic |
| | – | Temporální logika větvícího se času |
| CUDD | – | Colorado University Decision Diagram package |
| | – | Knihovna funkcí pro manipulaci s rozhodovacími diagramy |
| DNF | – | Disjunctive normal form |
| | – | Disjunktivní normální tvar |
| LTL | – | Linear temporal logic |
| | – | Lineární temporální logika |
| OBDD | – | Ordered binary decision diagram |
| | – | Uspořádaný binární rozhodovací diagram |
| ROBDD | – | Reduced ordered binary decision diagram |
| | – | Redukovaný uspořádaný binární rozhodovací diagram |
| SAT | – | Satisfialibity problem |
| | – | Problém splnitelnosti |
| SMV | – | Symbolic model verifer |
| | – | Nástroj pro modelování a verifikování systémů |
| TL | – | Temporal logic |
| | – | Temporální logika |
| UDNF | – | Úplná disjunktivní normální forma |

# Obsah

# Seznam obrázků

# 1 Úvod

Na moderní počítačové programy jsou stále více kladeny různé požadavky zabývající se správností a spolehlivostí programů. Pomocí těchto požadavků se snažíme vyvarovat či eliminovat více či méně závažným chybám, které jsou součástí počítačových programů. Nyní je obrovské množství programů součástí různých produktů a systémů, které ovlivňují, ulehčují naše bytí a někdy na těchto programech dokonce závisí i náš život. Z těchto a také z finančních důvodů je důležité, aby byl zajištěn co nejmenší počet chyb, či jejich eliminace.

Pro eliminaci a odhalení chyb existuje mnoho způsobů. Mezi nejjednodušší způsoby patří testování vytvořených programů. Testování jako takové představuje techniku, pomocí které vyzkoušíme konkrétní program pro určitá vstupní data. Stává se však, že programy jsou velice komplexní a složité. Pro otestování komplexních a složitých programů by bylo potřeba vytvořit a následně vyzkoušet tak obrovské množství vstupních dat, že by nám jejich zpracování zabralo velké množství času. Z tohoto důvodu není možné otestovat programy nad všemi možnými daty. Je tedy nutné vynechat některá vstupní data, či jejich kombinace. Tímto vynecháním se do testování nemusí zahrnout některá problémová místa, či chování systému, které se může následně negativně projevit. Pomocí testování tedy můžeme odhalovat chyby konkrétních programů. Nelze však s jistotou říci, že neexistují žádné chyby.

Ačkoliv testování nevyloučí chybu, tak se stále používá jako jedna z hlavních technik zjišťující správnost a spolehlivost programů. Díky nedostatku testování a nezaručení neexistence chyb byly postupně vynalézány další techniky pro zajištění bezchybnosti programů. Jednou z dalších technik byly pokusy dokázat správnost programů takovým způsobem, jako se děje v matematice dokazování pravdivosti matematických tvrzení. Tuto techniku bylo možné použít jen pro triviální programy. Pro programy, které byly více složitější, již tato technika nebyla použitelná. Nastala tedy potřeba nalézt techniku, která by byla někde mezi testováním a zmíněným dokazováním. Techniky, které byly požadované, měly být automatizovatelné, dále by měly odhalovat chyby a také určit, zda již v programu chyby dále nejsou. Tyto techniky však není možné realizovat, aniž bychom se neomezili na konkrétní situace.

Další vývoj technik zajišťujících správnost a spolehlivost programů se zaměřil na programy, které jsou konečně-stavové. Jedná se o programy, které mohou nabývat různých vnitřních stavů, které se v průběhu činnosti programu mohou různě měnit. Někdo by mohl namítat, že použití technik založených na konečně-stavových programech je velké omezení. Programy, které jsou však řešeny na digitálním zařízení, tuto vlastnost (vnitřní stavy) mají. Ukazuje se tedy, že je dobré použití programů, které jsou konečně-stavové. Tato technika se dále vyvíjela a zdokonalovala.

Již v polovině sedmdesátých let byly vytvořeny první návrhy deduktivních systémů, které byly určeny pro verifikaci konečně stavových programů neboli konečných modelů programů. V těchto deduktivních systémech se využívalo temporálních logik. Použití této logiky pro formulování tvrzení o programech znamenalo velký posun k automatizované verifikaci. Temporální logika totiž umožňuje vyjadřovat tvrzení o chování systémů.

Další krok ke zdokonalení přišel začátkem let osmdesátých. Byl objeven přístup, který nevyžadoval popsat jak program, tak i vlastnosti tohoto programu jako formule temporální logiky. Stačilo jen vytvořit konečně-stavový program, který modeloval konkrétní systém. V temporální logice se pak pouze popisovaly jen vlastnosti, jejichž pravdivost bylo třeba ověřit v daném modelu. Výpočetně se jednalo o podstatně jednodušší problém. Dále byla významná i skutečnost, že při ověřování modelu je možno v případě negativní odpovědi vytvořit odpovídající protipříklad.

Při určování správnosti a spolehlivosti programů se zjistilo, že je velmi důležité zohlednit velikost modelu. Počet stavů modelu totiž závisí exponenciálně na počtu proměnných, které jsou v programech obsaženy. Tato situace je pojmenována jako stavová exploze.

V letech následujících byla snaha o vyřešení problémů se stavovou explozí. Jedním ze způsobů, jak zvládnout velikost stavového prostoru, je použití metody symbolické reprezentace stavového prostoru. Usnadnění spočívá v tom, že algoritmus pracuje s množinami stavů, místo s jednotlivými daty. Symbolické ověřování modelů je založené na tom, že množiny stavů lze ekvivalentně zadat pomocí výrokových formulí. Obrovskou množinu stavů je tedy možné reprezentovat pomocí velmi malé formule. Z tohoto vyplývá, že úspěch verifikačního algoritmu závisí čistě na kompaktnosti symbolické reprezentace.

Následně byly vymýšleny další techniky a různá vylepšení určování správnosti a spolehlivosti programů. Chyby však v programech byly a budou i nadále. A to z toho důvodu, že programy, které jsou psány, vytváří člověk a nejsou vytvářeny automatizovaně, bez lidského zásahu. Dnešní metody, pokud jsou využívány, zmenšují počet chyb v programech a velice podstatným způsobem eliminují významné chyby. Proto je použití těchto metod a technik velice žádoucí. Podrobný vývoj technik a různých vylepšení určování správnosti a spolehlivosti programů je možno prostudovat v multioborovém semináři *Správnost počítačových programů - očekávání a realita* [1].

Cílem této práce bylo ukázat pomocí dvou nástrojů, Symbolic model verifer [26] a NuSMV [32], jednu z technik určování správnosti a spolehlivosti programů. Jedná se o techniku symbolického ověřování modelů. Ověřování neboli verifikace jako pojem takový, pochází z latinského verum facere, neboli činit pravdivým. Ověřování modelů tedy představuje verifikaci, která ověřuje, zda model systému splňuje zadaný požadavek či nikoliv. Technika pro určování správnosti a spolehlivosti programů nazvaná symbolické ověřování modelů využívá pro reprezentaci stavového prostoru struktury zvané binární rozhodovací diagramy. Na těchto strukturách jsou založeny oba nástroje, které jsou v této diplomové práci popsány. Nástroje Symbolic model verifer a NuSMV jsou tedy nástroje určené pro symbolické ověřování modelů. Symbolic model verifer a následná varianta NuSMV jsou nejpoužívanějšími prostředky pro verifikaci modelů převážně hardwarových systémů. Nástroj NuSMV vznikl zefektivněním, novou implementací a rozšířením nástroje Symbolic model verifer. Zefektivnění bylo provedeno i v rámci uživatelského rozhraní. Bylo vytvořeno grafické uživatelské rozhraní, které je pro uživatele příjemnější oproti původní příkazové řádce. Další a specifičtější informace k nástrojům Symbolic model verifer a NuSMV budou dále uvedeny v této práci.

Pro lepší pochopení možností obou nástrojů byly nastudovány a následně v této práci uvedeny teoretické základy, na kterých jsou oba nástroje postaveny. V rámci této práce jsou uvedeny základní informace z teorie grafů, výrokové logiky, temporálních logik a z dalších oblastí. S využitím těchto informací jsou následně uvedeny informace o technice ověřování modelů a o symbolickém ověřování modelů, na kterém jsou nástroje Symbolic model verifer a NuMSV postaveny. Poslední a nejdůležitější částí této diplomové práce je část závěrečná, která popisuje účel a možnosti nástroje Symbolic model verifer na konkrétních modelech různých systémů. V této závěrečné části jsou na jednotlivých příkladech zhodnoceny výhody a nevýhody použití tohoto nástroje.

Tato diplomová práce je tedy rozdělena do několika kapitol, které se zabývají různými oblastmi. V první kapitole jsou uvedeny základní pojmy z teorie grafů, výrokové logiky a některé další vybrané pojmy. Jak již bylo řečeno, tak nástroje Symbolic model verifer a NuSMV jsou založeny na strukturách nazvaných binární rozhodovací diagramy. Z tohoto důvodu je v této kapitole připomenuta definice grafů a jejich dělení. Neboť grafy, které představují binární rozhodovací diagramy, jsou v podstatě kořenové orientované acyklické grafy, které reprezentují logickou funkci. Tato kapitola je dále doplněna o informace z oblasti výrokové logiky. Jedná se o informace týkající se pravdivostních funkcí, konjunktivních, resp. disjunktivních normálních forem a dalších pojmů výrokové logiky.

Po této kapitole následuje kapitola nazvaná formální verifikace. Tato kapitole dává ucelený přehled o základních metodách formální verifikace, o logikách, ve kterých je při vytváření konkrétních modelů možno definovat verifikovaná kritéria a také poskytuje informace o nástrojích, které jsou určeny pro formální verifikaci. V této kapitole tedy nalezneme bližší informace o temporální logice, která je základem temporální logiky větvícího se času. Tato temporální logika větvícího se času je využívána jako prostředek pro vyjádření požadavků na verifikovaný model systému v nástrojích Symbolic model verifer a NuSMV. Pro ucelený přehled o těchto logikách je kapitola doplněna i o informace o lineární temporální logice. Tuto kapitolu uzavírají informace týkající se protipříkladu ve formální verifikaci a přehled nástrojů, které lze použít k verifikování v různých oblastech.

Nástroje Symbolic model verifer a NuSMV jsou určené pro symbolické ověřování modelů, tzv. symbolický model checking. Aby bylo možno dostatečně popsat symbolické ověřování modelů, je třeba nejprve uvést několik informací o ověřování modelu. Ověřování modelu neboli model checkingu je věnována následující kapitola. Kapitola poskytuje čtenáři informace o model checkingu a o základních rysech této techniky. Dále kapitola popisuje postup, kterým se ověřování modelů provádí a jak modelovat konkrétní systém.

Po kapitole popisující ověřování modelů následuje kapitola zaměřená na symbolické ověřování modelů. V případě symbolického ověřování modelů je využíváno pro reprezentaci stavového prostoru struktur nazvaných binární rozhodovací diagramy. Tato kapitola tedy seznamuje čtenáře s těmito strukturami a dalšími informacemi, které se dotýkají těchto diagramů. V kapitole jsou taktéž uvedeny informace o uspořádaných binárních rozhodovacích diagramech, které jsou vylepšením předešlých binárních rozhodovacích diagramů. Uvedené uspořádané binární rozhodovací diagramy mohou však obsahovat různé redundance. Jejich zamezením vznikly redukované uspořádané binární diagramy. Tyto diagramy jsou v této kapitole taktéž popsány. Tuto kapitolu uzavírá popis operací,

které je možné nad těmito strukturami provádět a způsob reprezentace těchto struktur pomocí různých knihoven.

V kapitole o formální verifikaci byl uveden přehled nástrojů, které je možné využít v různých oblastech. Předposlední kapitola podrobněji popisuje nástroje, které jsou založené na strukturách uvedených v předešlé kapitole. Jedná se hlavně o tyto nástroje: Symbolic model verifer, Cadence SMV a NuSMV.

Závěrečnou kapitolu tvoří příkladová studie, ve které jsou pomocí několika modelů konkrétních systémů ukázány možnosti a omezení nástroje SMV. Jsou zde popsány, vytvořeny a verifikovány modely představující konečně stavové systémy. Jedná se o modely automatů, kuchyňského spotřebiče, hardwarového obvodu, jednoduché křižovatky a většího modelu přechodu pro chodce. Tato kapitola shrnuje předešlé informace a prakticky ukazuje možnosti verifikace pomocí uvedených nástrojů.

# 2 Vybrané pojmy z teorie grafů a výrokové logiky

V této kapitole si připomeneme základní pojmy týkající se teorie grafů, dále výrokové logiky a speciálních témat, jako jsou konjunktivní, resp. disjunktivní normální formy a význam Shannonovy expanze.

## 2.1 Grafy

Jedním z cílů této práce bylo popsat binární rozhodovací diagramy (BDD), uspořádané binární rozhodovací diagramy (OBDD), případně redukované uspořádané binární rozhodovací diagramy (ROBDD). Všechny tyto diagramy je možno znázornit pomocí struktur, kterým se obecně říká *graf*. Informace uvedené v této kapitole jsem čerpal z online přednášek předmětu *Diskrétní matematika* [13].

Grafem můžeme nazvat strukturu, která je tvořená dvojicí z množiny pojmenované E a neprázdnou množinou pojmenovanou V. Prvky množiny E se nazývají hrany grafu a mohou to být uspořádané, případně neuspořádané dvojice prvků. Prvky množiny V se nazývají vrcholy grafu. Pro tyto vrcholy se taktéž používá pojmenování uzly.

**Definice 2.1 Graf**
*Často také nazýván obyčejný, či jednoduchý neorientovaný graf:*

- *je uspořádaná dvojice G = (V, E),*
- *kde V je množina vrcholů,*
- *E je množina hran - množina vybraných dvouprvkových podmnožin množiny vrcholů.*

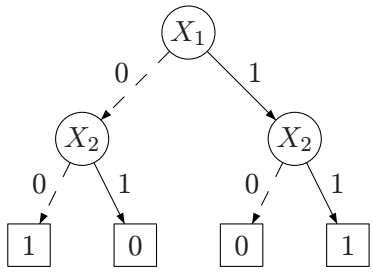
Grafy se mohou využívat a slouží jako různé abstrakce mnoha problémů. Používají se jako zjednodušující model reálné sítě. Jako příklad můžeme uvést dopravní, počítačové sítě a mnoho dalších. Modely sítí zdůrazňují topologické vlastnosti vrcholů, které představují konkrétní reálné objekty, tedy jejich vzájemné propojení.

### 2.1.1 Dělení grafů

- Podle četnosti hran rozdělujeme grafy na jednoduché a multigrafy. V případě jednoduchého grafu vede mezi libovolnými dvěma vrcholy nejvýše jedna hrana. V druhém případě, kdy se jedná o multigraf, může vést mezi dvěma vrcholy libovolný počet hran.
- Podle četnosti se mohou grafy rozlišovat ještě takto: na grafy řídké a grafy husté. Grafy řídké jsou grafy, s relativně malým množstvím hran vzhledem k počtu vrcholů. Husté grafy opačně, tedy s velkým množstvím hran vzhledem k počtu vrcholů.
- Dále grafy dělíme podle orientace hran na orientované a neorientované. Orientované grafy jsou takové, jejichž hrany jsou uspořádané dvojice vrcholů. Grafy neorientované jsou takové, jejichž hrany jsou dvouprvkové množiny vrcholů.

- Podle souvislosti rozlišujeme grafy na souvislé, u kterých existuje cesta mezi každou dvojicí vrcholů a na nesouvislé, kde platí opak.
- Grafy můžeme dělit i podle existence v grafu. Grafy pak pojmenováváme jako cyklické a acyklické. Jako příklad acyklických grafů je možno zmínit například stromy.
- Jedním z důležitých rozdělování grafů je i podle toho, zda lze graf nakreslit do roviny tak, aniž by se křížily jednotlivé hrany. Rozdělujeme tedy grafy na rovinné a nerovinné.
- existuje ještě několik způsobů dělení grafů. Jedním z nich je například dělení na ohodnocené a neohodnocené grafy.

Grafy, které představují binární rozhodovací diagramy (BDD) jsou v podstatě kořenové orientované acyklické grafy reprezentující logickou funkci. Takový graf můžeme vidět na obrázku č. 1.




Obrázek 1: BDD diagram

## 2.2 Výroková logika

V této diplomové práci se budeme následně zabývat symbolickým ověřováním modelu. Symbolické ověřování modelu je založeno na tom, že konkrétní množiny stavů lze ekvivalentně zapsat pomocí výrokových formulí, tedy formulí výrokové logiky. Pro vypracování následující části byly nastudovány informace z online skripta *Matematické logiky* [14].

Výroková logika označuje formální odvozovací systém, ve kterém atomické formule tvoří výrokové proměnné. Základem výrokové logiky jsou výroky. Výrok jako takový nedefinujeme. Jen určíme, co si pod pojmem výrok budeme představovat. Za výrok tedy budeme považovat každé tvrzení, jakoukoliv oznamovací větu, o které se dá říci, zda je pravdivé či nikoliv. Tedy můžeme určit pravdivostní hodnotu tohoto výroku.

Výroky, mohu být pravdivé i nepravdivé. Mezi výroky například nepatří tázací věty nebo věty, u kterých nelze jednoznačně určit její pravdivostní hodnotu.

Výroky je taktéž možné znegovat. Při používání výroků se pracuje s logickými spojkami, které se zapisují pomocí funktorů: konjunkce, disjunkce, implikace a ekvivalence. Tyto spojky spojují jednotlivé výroky do složených výroků.

**Definice 2.2 Jazyk výrokové logiky**

*Abeceda jazyka výrokové logiky je množina následujících symbolů:*

- *Výrokové symboly (o, p, q, ...)*
- *Symboly logických spojek - funktorů (¬, ∧, ∨, ⇒, ⇔)*
- *Pomocné symboly: (, ), [, ], {, }*

**Definice 2.3 Pravdivostní funkce**

*Pravdivostní funkcí f($p_1, p_2, p_3, ..., p_n$) proměnných $p_1,\ p_2,\ p_3,\ \dots,\ p_n$ rozumíme zobrazení* $\{0,\ 1\}^n \rightarrow \{0, 1\}$.

- *Každá kombinace ohodnocení n vstupních proměnných (0, 1) generuje výstupní hodnotu 0 nebo 1.*

### 2.2.1 Pravdivostní funkce

#### 2.2.1.1 Negace

| X | ¬ X |
|---|---|
| 0 | 1 |
| 1 | 0 |

Logická negace je unární logická operace, jejíž hodnota je *nepravda*, právě když první vstupní hodnota je *pravda* a naopak.

#### 2.2.1.2 Konjunkce

| X | Y | X ∧ Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

X ∧ Y - je konjunkce výroků a můžeme číst „výrok X a zároveň výrok Y“.

#### 2.2.1.3 Disjunkce

| X | Y | X ∨ Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

X ∨ Y - je disjunkce výroků a můžeme „číst výrok X nebo výrok Y“.

#### 2.2.1.4 Implikace

| X | Y | X ⇒ Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

X ⇒ Y - je implikace a můžeme číst „jestliže výrok X, pak výrok Y“.

#### 2.2.1.5 Ekvivalence

| X | Y | X ⇔ Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

X ⇔ Y - je ekvivalence a můžeme číst „výrok X právě tehdy když výrok Y“.

## 2.3 Konjunktivní, Disjunktivní normální forma

V této části si popíšeme pomocí definic Booleovu funkci a dva možné způsoby vyjádření funkcí skrze konjunktivní normální formu, neboli CNF a skrze disjunktivní normální formu, což je naopak DNF.

Literálem nazveme logickou proměnnou nebo negaci logické proměnné.

**Definice 2.4 Booleova funkce**

*Booleovou funkcí n proměnných, kde n představuje přirozené číslo, rozumíme každé zobrazení* $f : \{0,\ 1\}^n \rightarrow \{0,\ 1\}$*, tj. zobrazení, které každé n-tici* $(x_1,\ x_2,\ x_3,\ \ldots\ ,\ x_n)$ *nul a jedniček přiřazuje nulu nebo jedničku, označenou f*$(x_1,\ x_2,\ x_3,\ \ldots\ ,\ x_n)$*.*

**Definice 2.5 Konjunktivní normální forma - CNF**

*Řekněme, že formule je v konjunktivním normálním tvaru, jestliže je konjunkcí jedné nebo několika formulí, z nichž každá je literálem nebo disjunkcí literálů.*

- *Literál a disjunkce literálů se taktéž pojmenovávají* ***maxterm*** *nebo* ***klausule****.*
- *Pokud každá klausule obsahuje všechny proměnné, říkáme, že se jedná o úplný konjunktivní normální tvar - UCNF.*
- *Obecně CNF:* $(p_{11} \vee p_{21} \vee \ldots \vee p_{n1}) \wedge (p_{12} \vee p_{22} \vee \ldots \vee p_{n2}) \wedge \ldots \wedge (p_{1m} \vee p_{2m} \vee \ldots \vee p_{nm})$

**Definice 2.6 Disjunktivní normální forma - DNF**

*Řekněme, že formule je v disjunktivním normálním tvaru, jestliže je disjunkcí jedné nebo několika formulí, z nichž každá je literálem nebo konjunkcí literálů.*

- *Literál a konjunkce literálů se v tomto případě pojmenovávají* ***minterm****.*
- *Pokud každý minterm obsahuje všechny proměnné, říkáme, že se jedná o úplný disjunktivní normální tvar - UDNF.*
- *Obecně CNF:* $(p_{11} \wedge p_{21} \wedge ... \wedge p_{n1}) \vee (p_{12} \wedge p_{22} \wedge ... \wedge p_{n2}) \vee ... \vee (p_{1m} \wedge p_{2m} \wedge ... \wedge p_{nm})$

## 2.4 Shannonova expanze

Informace uvedené v následující části, která popisuje Shannonovu expanzi, byly čerpány z bakalářské práce *Řešení problému splnitelnosti booleovské formule (SAT) pomocí binárních rozhodovacích diagramů (BDD)* [2].

Shannonova expanze neboli Shannonův expanzní teorém je definován následovně:

$$f(x) = \neg x_i \cdot f(x_1, ..., x_i = 0, ..., x_n) \vee x_i \cdot f(x_1, ..., x_i = 1, ..., x_n)$$

Funkce $f(x_1, ..., x_i = 0, ..., x_n)$ je kofaktorem funkce *f*, který je rozložený podle proměnné $x_i$. Kofaktor se taktéž někdy nazývá restrikce. Jedná se o výsledek funkce po dosazení konstantní hodnoty za proměnnou $x_i$ funkce *f*. Nahradíme tedy všechny výskyty proměnné $x_i$ logickou jedničkou, a všechny výskyty proměnné $\neg x_i$ logickou nulou. Kofaktor lze zapsat i jednodušším způsobem takto: $f|_{x \leftarrow k}$. Pokud použijeme tento jednodušší způsob, tak zápis jakékoliv funkce *f* bude vypadat následovně:

$$f(x) = \neg x \cdot f|_{x \leftarrow 0} + x \cdot f|_{x \leftarrow 1}$$

V případě, že bychom chtěli vyjádřit další operace, tak to provedeme pomocí spojení operace restrikce s dalšími algebraickými operacemi. Například může vyjádřit operaci kompozice. V tomto případě při kompozici funkce $g$ nahrazuje proměnnou $x$ funkce *f*. Zápis kompozice bude vypadat následovně:

$$f|_{x \leftarrow g} = \neg g \cdot f|_{x \leftarrow 0} + g \cdot f|_{x \leftarrow 1}$$

Shannonovu expanzi můžeme tedy využít pro aplikaci libovolného binárního logického operátoru. Touto metodou tedy můžeme rozkládat booleovské funkce na součet dvou „podfunkcí“ původní funkce.

**Příklad:** rozklad funkce F = AB + BC + AC

F = AB + BC + AC = AB(C + C′) + BC(A + A′) + AC(B + B′) = ABC + ABC′ + BCA + BCA′ + ACB + ACB′ = ABC + ABC′ + ABC + A′BC + ABC + AB′C = ABC + A′BC + AB′C + ABC′ = A(BC + BC′) + A′(BC + B′C)

Při řešení se využilo této vlastnosti: *(X + X′) = 1*.

# 3 Formální verifikace

Tato kapitola dává přehled o základních metodách formální verifikace, popisuje logiku, která se využívá při specifikování verifikovaných kritérií modelů systémů a na závěr předkládá stručný přehled nástrojů pro formální verifikaci. Tato kapitola byla vypracována na základě nastudovaných informací z bakalářské práce: *Práce s verifikačním nástrojem* [5], z online dokumentu *Linear Temporal Logic* [15] a z online prezentace *Validace a verifikace* [16].

V oblasti ověřování systémů se v dnešní době využívají moderní metody verifikace a formální analýzy. Verifikace je v podstatě ověřování a kontrola pravdivosti různých výroků, argumentů, hypotéz, logických systémů konfrontací s pomocí faktů nebo ověřování platnosti nějakého úsudku formální analýzou. Existují dva typy verifikací. Prvním je experimentální verifikace. Experimentální verifikace je v podstatě soubor operací, které aby dokázaly platnost hypotézy, srovnávají její důsledky se zkušenostmi. Naproti tomu formální verifikace je postup, který pomocí logických operací ověřuje shodu s přijatými axiomy. Tam, kde se používají počítačové systémy, formální verifikace vyvrací nebo dokazuje správnost daného systému vzhledem k formální specifikaci nebo vlastnosti za použití formálních matematických metod.

Existují tři základní metody formální verifikace:

**Equivalence checking,** neboli také ověřování ekvivalencí, rozhoduje, zda je specifikace systému ekvivalentní s daným systémem vzhledem k danému druhu ekvivalence v chování.

**Model checking,** neboli také ověřování modelů rozhoduje, zda systém splňuje danou vlastnost, kritérium, či nikoliv. V případě, že systém konkrétní vlastnost, kritérium nesplňuje, dokáže model checking ukázat tzv. protipříklad, tedy chování systému, které porušuje danou vlastnost.

**Theorem proving,** neboli také dokazování vět, představuje následující. Systém i jeho vlastnosti jsou vyjádřeny jako formule v určitém systému matematické logiky, a námi zmíněný theorem proving hledá důkaz vlastnosti.

## 3.1 Temporální logika

Jedním z cílů této diplomové práce bylo ukázat vytváření modelu systému a jeho ověřování pomocí nástroje Symbolic model verifer, SMV. Při vytváření konkrétních modelů je možno specifikovat resp. definovat verifikovaná kritéria, vlastnosti, pomocí temporální logiky větvícího se času (Computation Tree Logic, CTL). Pro dostatečné pochopení CTL je třeba nejprve uvést něco o temporálních logikách jako takových.

Temporální logika (Temporal Logic, TL) je rozšíření výrokové logiky o modální operátory, které dovolují specifikovat vlastnosti týkající se času. Jedná se tedy o formalizmus, který popisuje posloupnosti přechodů mezi stavy systému. Čas zde však není myšlen explicitně. Pomocí formule můžeme totiž specifikovat, že někdy může být nějaký stav

dosažitelný, nebo že bychom nikdy nevstoupili do nějakého chybového stavu. Lze vyjádřit, že daná vlastnost musí pořád platit od jistého okamžiku po celou dobu výpočtu neboli běhu programu. To znamená, že výrok nemusí být pouze staticky pravdivý nebo nepravdivý. Pravdivost výroků je tedy v temporální logice závislá na čase, resp. na aktuální pozici ve stavovém prostoru a jejím okolí. Vlastnosti jako někdy a nikdy jsou specifikovány speciálními temporálními operátory. Tyto operátory mohou být různě kombinovány s booleovskými spojkami nebo vzájemně vnořeny.

Jednotlivé podmnožiny temporální logiky se od sebe liší v operátorech, které je možné použít, a sémantice používaných operátorů. Kromě klasických čtyř operátorů výrokové logiky $\{\wedge, \vee, \neg, \rightarrow\}$ zahrnují různé varianty TL minimálně tyto následující *modální operátory*:

1. **X** p, neboli „následně p“ - musí být zaručeno, že vlastnost p bude splněna v následujícím stavu

2. **F** p, neboli „někdy v budoucnu bude platit p“ - tento operátor se používá k určení, že vlastnost označená p bude splněna v některém ze stavu cesty

3. **G** p, neboli „vždy p“, nebo „globálně p“ - vlastnost označená p je splněná v každém stavu cesty

4. p **U** q, neboli „p dokud neplatí q“ - jedná se o operátor, který používá dvou vlastností. První je, že formule platí, pokud na cestě existuje nějaký stav, kde je splněna vlastnost q a v každém předcházejícím stavu je splněna vlastnost p

5. p **R** q, neboli „p uvolňuje q“ - tento operátor vyžaduje, aby vlastnost q byla splněna ve všech stavech cesty až po první stav včetně, ve které formuli platí p. Není však požadováno, aby vlastnost p byla vůbec někdy splněna.

## 3.2 Temporální logika větvícího se času

Zvláštním případem temporální logiky je temporální logika větvícího se času, anglický název je Computation tree logic, neboli CTL. Tato logika je využívána jako prostředek pro vyjádření požadavků na verifikovaný systém například v nástroji SMV, či v nástroji NuMSV. Computation tree logic je temporální logikou využívající branching-time.

V CTL formulích se mohou používat kvantifikátory a temporální operátory. Kvantifikátory se používají k popisu větvení struktury ve stromech. V CTL existují dva kvantifikátory. Prvním z nich je $A$, který má význam „pro všechny cesty“ a druhým je $E$, který má význam „pro některou cestu“. V konkrétních stavech se tyto kvantifikátory používají k určování toho, zda splňují nějakou vlastnost všechny nebo některé z cest vycházející z daného stavu. Temporální operátory popisují vlastnosti cesty pomocí stromu. CTL formule můžeme definovat například induktivně pomocí Backus-Naurovy formy:

$$\varphi ::= \bot|T|p|(\neg\varphi)|\varphi \wedge \varphi|\varphi \vee \varphi|\varphi \Rightarrow \varphi|AX\varphi|EX\varphi|A[\varphi U\varphi]|E[\varphi U\varphi]|AG\varphi|EG\varphi|$$
$$AF\varphi|Ef\varphi,$$

kde p je atomická formule a T, $\perp$ jsou symboly pro tautologii, resp. kontradikci. Operátory času ani operátory větví (A a E) se podle definice nemohou v Computation tree logic formuli nikdy vyskytovat samostatně. Musí se vyskytovat vždy ve dvojici. Časovému operátoru musí vždy předcházet operátor větví. CTL nemá žádné operátory pro minulý děj, minulost.

CTL používá pět základních operátorů (viz kapitola č. 3.1 o temporální logice), kterým musí vždy předcházet jeden z kvantifikátorů určených pro cesty. Kvantifikátory ($A$, $E$), které rozšiřují temporální logiku, můžeme vidět dále.

1. Unární $A\varphi$ – operátor „all“, výrok $\varphi$ platí ve všech posloupnostech stavů, které začínají v aktuálním stavu,

2. Unární $E\varphi$ – operátor „exists“, výrok $\varphi$ platí v některé z posloupností stavů, která začíná v aktuálním stavu,

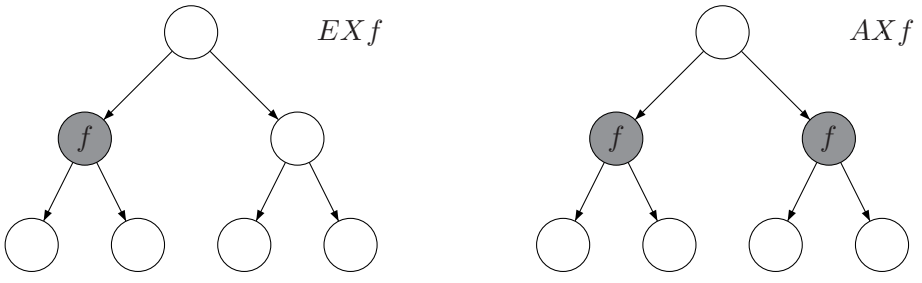
### 3.2.1 Příklady CTL

V nástroji SMV budeme vyjadřovat požadavky na verifikovaný systém. Je tedy dobré ukázat, jak vypadá konkrétní specifikace v tomto nástroji. Pro rozlišení CTL formule se v SMV používá slovo *SPEC*, které uvozuje začátek CTL formule. Konkrétní formule pak vypadá následovně:

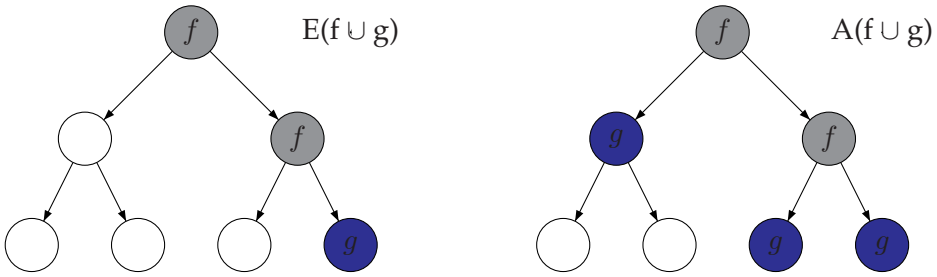
```
SPEC EG!(semafor1.barva = zelena & semafor2.barva = zelena)
```

CTL formule *EG!(semafor1.barva...* nám říká, že v každém z následujících stavů nebude existovat situace, kdy na semaforu pro chodce (semafor1) a na semaforu pro vozidla (semafor2) svítí zároveň barva zelená.

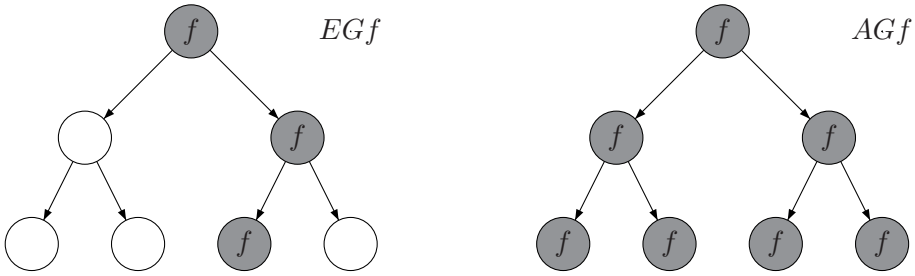
V grafické podobě můžeme sémantiku kvantifikátorů a temporálních operátorů vidět na následujících obrázcích. Na obrázku č. 2 je zobrazeno EXf a AXf. Na dalším obrázku č. 3 je zobrazeno E(f $\cup$ g) a A(f $\cup$ g). Poté je zobrazeno EGf a AGf na obrázku č. 4. Na posledním obrázku č. 5 je zobrazeno EFf a AFf.



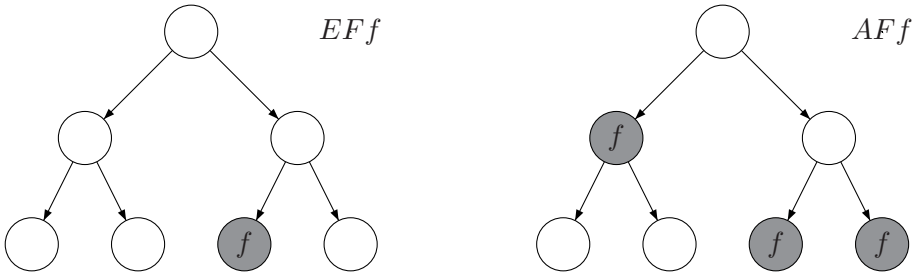

Obrázek 2: Sémantika kvantifikátorů a temporálních operátorů - EXf, AXf.




Obrázek 3: Sémantika kvantifikátorů a temporálních operátorů - E(f ∪ g), A(f ∪ g).




Obrázek 4: Sémantika kvantifikátorů a temporálních operátorů - EGf, AGf).




Obrázek 5: Sémantika kvantifikátorů a temporálních operátorů - EFf, AFf).

## 3.3 Lineární temporální logika

Jednou z dalších modálních temporálních logik (logik s vyjadřovací schopností ohledně časových souvislostí) je lineární temporální logika (Linear Temporal Logic, LTL). Lineární temporální logika nemá kvantifikátory cest $A$ nebo $E$. V lineární temporální logice je možné vyjádřit různé vlastnosti systému. Jako příklad můžeme uvést, že daná vlastnost musí platit od jistého okamžiku po celou dobu výpočtu neboli běhu programu. Množina všech formulí této logiky je definována následujícím zápisem:

$$\varphi ::= Q|\neg\varphi|\varphi_1 \wedge \varphi_2|Op(\varphi_1, ..., \varphi_n),$$

kde:

- Op je n-ární temporální operátor,
- Q je spočetná množina atomických propozic.

Temporálních operátorů pro LTL logiku existuje několik. Nejčastěji jsou však využívány dva operátory: binární until, označován písmenem $U$ a unární next, označován písmenem $X$.

### 3.3.1 LTL neformálně:

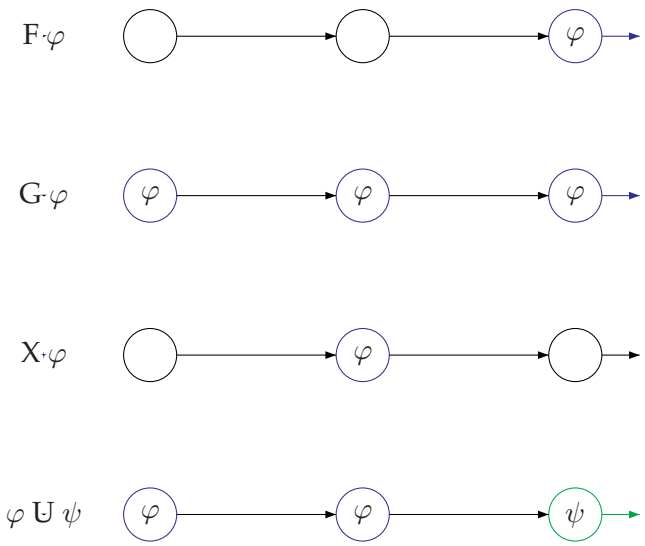
Formule $\varphi$:

- Vyhodnocuje se nad jedním během systému.
- Vyjadřuje se o platnosti základních propozic ve stavech konkrétního běhu.

### 3.3.2 Temporální operátory LTL:

| | | |
|---|---|---|
| F $\varphi$ | - (Future) | Někde v běhu platí $\varphi$. |
| G $\varphi$ | - (Globally) | V daném běhu vždy platí $\varphi$. |
| $\varphi$ U $\psi$ | - (Until) | Někde platí $\psi$ a do té doby platí $\varphi$. |
| X $\varphi$ | - (Next) | V příštím stavu platí $\varphi$. |
| $\varphi$ W $\psi$ | - (Weak Until) | Jako Until, ale $\psi$ nemusí nastat. |
| $\varphi$ R $\psi$ | - (Release) | $\psi$ platí, dokud neplatí $\varphi \wedge \psi$. |

Zmíněné temporální operátory LTL jsou zobrazeny v přehlednější podobě, grafické, na obrázku č. 6.




Obrázek 6: Temporální operátory LTL graficky.

## 3.4 Protipříklad

Obecně můžeme protipříkladem nazvat výjimku z nějakého uvažovaného obecného pravidla. Jedná se tedy o speciální případ nepravdivosti formule s univerzálním kvantifikátorem. Tímto univerzálním kvantifikátorem může být například formule typu "Pro všechny ".

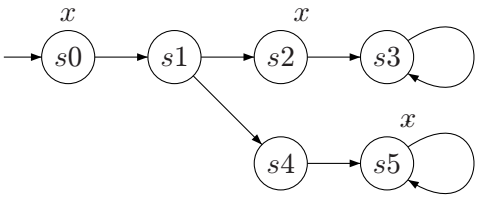
### 3.4.1 Protipříklad ve formální verifikaci

Ve formální verifikaci je protipříklad určitý běh systému, neboli chování toho systému, které porušuje námi ověřovanou vlastnost. V případě, že protipříklad nalezneme, tak se námi ověřovaný systém upraví tak, aby se v příštím průchodu dokázalo, že systém již naši konkrétní vlastnost splňuje, nebo je nalezen protipříklad nový. Po provedení několika takovýchto kroků je možno vytvořit lepší systém.

Pro praktickou ukázku jak bude vypadat protipříklad v nástroji SMV, uvažujme námi zvolený libovolný automat, který je zobrazen na obrázku č. 7.

Nyní si nebudeme popisovat, jak tento zvolený automat převedeme do SMV. Víme však, že vlastnosti, které požadujeme verifikovat, se zapisují způsobem, který je ukázán níže.

```
SPEC EG x
```




Obrázek 7: Automat.

Pomocí slova *SPEC* definujeme, že se jedná o specifikaci vlastností. Dále jsou uvedeny operátory CTL logiky a vlastnost *x*, představující určité chování automatu. Uvedenou specifikací ověřujeme, zda existuje cesta, kde stále platí uvedený stav *x*. Nástroj SMV nám poskytne v tomto případě negativní odpověď a protipříklad, na kterém je ukázáno chování systému, které porušuje námi ověřovanou vlastnost. Verifikovaná vlastnost nemůže být pravdivá, neboť hned druhý stav vlastnost *SPEC EG x* nesplňuje. Výpis, který nám poskytne nástroj SMV je následující:

```
-- specification EG x is false
-- as demonstrated by the following execution sequence
state 1.1:
x = 1
state = s0
```

## 3.5 Verifikační nástroje

Tuto část zabývající se verifikačními nástroji jsem vypracoval díky informacím, které jsou uvedené v bakalářské práci: *Práce s verifikačním nástrojem - UPPAAL* [5].

Pro formální verifikaci existuje nepřeberné množství nástrojů. Velké množství těchto nástrojů je experimentálních, vznikají na různých univerzitách po celém světě v rámci projektů a diplomových prací. Jen malé množství těchto nástrojů je vytvářeno v rámci větších projektů a je určeno pro komerční využití a nasazení pro testování komplexnějších systémů. Tyto nástroje jsou zaměřeny na jednotlivé typy verifikací. Pomocí těchto nástrojů jsme schopni provádět kontrolu modelů, kontrolu ekvivalence i dokazování vět. Většina nástrojů umožňuje provádět ne jeden typ verifikace, ale často se zaměřují na širší využití. Jeden z nástrojů určených pro širší využití je například CadenceSMV, který se zaměřuje na ověření kontroly modelů, tak i na dokazování vět, viz uvedený seznam níže.

Velký přehled nástrojů sloužících pro verifikaci je umístěn na internetu v databázi verifikačních nástrojů - *YAHODA* [24]. Tato databáze poskytuje ucelený přehled dostupných nástrojů pro verifikaci. Je možné zde najít základní informace o jednotlivých nástrojích. Dozvíme se zde, zda jednotlivé verifikační nástroje zahrnují některý z typů kontrol: kontrola ekvivalence, kontrola modelů a dokazování vět. Dále se dozvíme, zda verifikační nástroje mají grafické uživatelské rozhraní, zdali jsou komerční, volně šiřitelné a na jakých platformách je možné verifikační nástroje užívat.

- Mezi známé verifikační nástroje z oblasti ověření kontroly ekvivalence patří například: CADP, CWB - NC, DREAM, Mocha, Moped a mnoho jim podobných.
- Do skupiny nástrojů z oblasti ověření kontroly modelů, můžeme uvést například tyto nástroje: Bandera, Blast, CadenceSMV, HyTech, NuSMV, TIMES, TRON, UPPAAL. Jako v předešlém výběru i tento seznam nástrojů je mnohem delší než zde uvádím.
- Do poslední skupiny, zabývající se dokazováním vět, můžeme uvést tyto nástroje: Atelier B, Cadence SMV, Expander2, IOA Toolkit, TPS a některé další nástroje.

V kapitole č. 6 budou popsány nástroje, které jsou založené na binárních rozhodovacích diagramech. Jedná se o nástroje Symbolic model verifer (SMV), Cadence SMV a NuSMV.

# 4 Model checking

Tuto část zabývající se model checkingem jsem vypracoval s využitím informací z diplomové práce *Verifikace s SMV* [4], dále z bakalářské práce *Práce s verifikačním nástrojem* [5], ze skript *Teorie automatů a formálních jazyků* [3], také z diplomové práce *Porovnání softwarových nástrojů pro model checking* [11] a s pomocí prezentace *Symbolický přístup k metodě ověřování modelu* [23].

Správnost a spolehlivost jsou základní požadavky, které na všechny systémy klademe. Ve skutečnosti existuje mnoho systémů, ve kterých jsou menší či větší chyby. Mnoho z těchto chyb je časem odhaleno a opraveno, některé jsou v produktu nadále a my se s nimi naučíme žít. Z tohoto a dalších důvodů je problematika správnosti a spolehlivosti systémů, tj. výzkum a vývoj metod, které by snížily počet chyb v systémech, jedním z hlavních témat oboru verifikace.

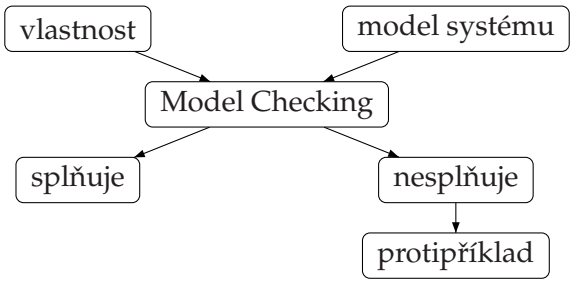
Se stále větším nástupem počítačových a řídicích systémů se objevuje důležitost možnosti ověření správného návrhu těchto systémů. Prvořadým úkolem je testování výsledného produktu, dalším způsobem ověřování jsou formální verifikační metody.

Nejčastěji používaný způsob, jak odhalit chyby v jednotlivých systémech, je testování těchto systémů. Testování je vyzkoušení systému pro několik typických situací, nebo pro omezené množství různých vstupních posloupností. Vzniká však problém v tom, že systémy jsou někdy velmi komplexní objekty. Množství vlastností, které by bylo žádoucí vyzkoušet, je tedy velmi velké. Zpracování těchto vlastností by mohlo v některých případech trvat neúnosnou dobu. Neprovádí se tedy kompletní testování systému, ale dochází k vynechávání některých vlastností. Tímto může dojít k případnému vynechání některých, pro reálný systém podstatných vlastností. Testování může odhalit chyby v systému, ale nemůže garantovat jejich naprostou neexistenci. Pokud dojdeme ke kladnému výsledku při testování, tak nemáme zaručeno, že bude příznivý i pro situace, které jsme netestovali.

Na druhou stranu, způsob řešení zahrnující verifikační techniky ověřuje chování a vlastnosti systému za všech okolností. Tento způsob řešení se začal používat o mnoho let později než jednoduché testování. Jednalo se o využití přístupu, který nevyžadoval popsání modelů i jeho vlastností formulemi temporální logiky. Stačilo, pokud byl vytvořen model s konečným počtem stavů modelující uvažovaný systém a v temporální logice stačilo pouze popsat vlastnosti, jejíž pravdivost v tomto modelu jsme chtěli ověřovat. Tento způsob řešení je nazýván model checking, neboli technika ověřování modelů.

Model checking představuje verifikaci, která ověřuje, zda model systému splňuje zadaný požadavek. Verifikace probíhá automaticky podle předem daného algoritmu. Princip model checkingu je znázorněn na obrázku č. 8. Mezi základní rysy model checking můžeme uvést tyto:

- Není ověřován přímo samotný systém, nýbrž jeho model, který je abstrakcí daného systému.
- Není možné postihnout všechny naše požadavky, neboť požadavky na systém jsou zadávány v temporální logice.




Obrázek 8: Princip model checkingu.

- Časová složitost samotné verifikace je závislá na velikosti verifikovaného modelu. Při výběru modelu musíme zvolit kompromis. V případě, že máme komplexnější model, který systém úplně popíše, může být problém určení platnosti formule nerozhodnutelný. Pokud však zvolíme jednoduchý model, je rozhodování platnosti formule rychlé. Na druhou stranu však nejsme schopni v něm podchytit přesně verifikovaný systém, ale jen částečně jeho abstrakci. Nebot'každým zjednodušením modelu proti systému můžeme vynechat část, která může způsobit chybu. Z tohoto důvodu nemáme zaručenou správnost ani po verifikaci.
- Při ověřování našeho modelu jsme schopni v případě negativní odpovědi nalézt i odpovídající protipříklad.

Postup, kterým se provádí model checking se skládá ze tří částí:

**Vlastní modelování** V případě modelování musí uživatel přepsat systém, nad kterým by chtěl dále verifikovat jeho vlastnosti, do určitého matematického nebo jiného formalismu. V našem případě budeme uvažovat přepis do modelovacího jazyka verifikačního nástroje SMV.

**Specifikace vlastností** Specifikací vlastností je myšleno jejich definování. Definujeme tedy vlastnosti zkoumaného systému, které chceme následně ověřovat, neboli verifikovat. Vlastnosti, které chceme ověřovat, mohou být například uváznutí (deadlock), živost systému, či ověření invariant. Pro definování specifikací modelu se používají formule temporální logiky (LTL, CTL viz kapitola č. 3.2).

**Verifikace vlastností** Verifikace vlastností je plně automatický proces, při kterém se ověřují vlastnosti vytvořené v předchozím kroku. Pokud konkrétní vlastnost platí, pak je odpovědí *ANO*. Pokud však vlastnost neplatí, tak je jako odpověď vytvořen protipříklad (viz kapitola č. 3.4).

## 4.1 Modelování systému

V případě že chceme provádět formální verifikaci, tak musíme nějakým způsobem reprezentovat verifikovaný systém. K této reprezentaci slouží modely systému. Přirozenou formou těchto modelů systému je orientovaný graf. Vrcholy orientovaného grafu představují stavy modelovaného systému. V konkrétním stavu mohou být zachyceny vlastnosti systému jako obsah proměnných, alokované prostředky a další. Naopak hrany orientovaného grafu představují přechod systému z jednoho stavu do jiného možného stavu. Hrany tedy představují změnu sledovaných vlastností. K reprezentaci modelu systému je možno využít různé formalismy. Těmito formalismy mohou být například Kripkeho struktury, ohodnocené přechodové stavy a další.

### 4.1.1 Kripkeho struktura

Kripkeho struktura je nedeterministický automat s konečným počtem stavů.

**Definice 4.1 Kripkeho struktura**
*AP je množina atomických výroků. Kripkeho struktura je matice* $< S, S_0, R, L >$,

- *kde S je množina stavů,*
- $S_0 \subset S$ *je množina počátečních stavů,*
- $R \subseteq SxS$ *je totální relace,*
- *L je zobrazení z množiny formulí na jednotlivé stavy L:S*$\rightarrow 2^{AP}$.

# 5 Symbolický model checking

Symbolický model checking pro konečné stavové systémy využívá pro reprezentaci stavového prostoru (při ohodnocování splnitelnosti jednotlivých formulí) struktury zvané BDD (Binary decision diagram). Jedná se tedy o verifikační techniku, která pracuje s množinami stavů namísto práce s jednotlivými stavy. Abychom pochopili symbolické metody formální verifikace, tak je nejdříve nutné pochopit konkrétní struktury binárních rozhodovacích diagramů (BDD) a uspořádaných binárních rozhodovacích diagramů (OBDD).

## 5.1 Binární rozhodovací diagramy

Tuto část zabývající se binárními rozhodovacími diagramy jsem vypracoval s využitím informací získaných z *An Introduction to Binary Decision Diagrams* [6], *System and Software Verification: Model-Checking Techniques and Tools* [7], dále z informací uvedených v *Detekce tautologie pomocí BDD* [8], *Minimalizace neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů* [9], *Port programového balíku CUDD pod platformou Windows* [10], *EDA, verification, BDDs* [22] a také informací uvedených zde: *Porovnání dostupných programových balíků pro manipulaci s binárními rozhodovacími diagramy* [12]. Velkým přínosem pro mě bylo i studium dokumentů uvedených na stránkách *Introduction to Software Verification* [17].

V dnešní době se v praxi setkáváme s potřebou řešit složité problémy při návrhu nebo optimalizaci různých systémů, dále při testování a verifikování. Mnoho takových problémů se dá převést na problém nad nějakou doménou logických funkcí za použití malého počtu operací. Logické funkce se dají vyjádřit mnoha různými způsoby, například pravdivostními tabulkami, konjunktivními normálními formami (KNF), či disjunktivními normálními formami (DNF), mapou a dalšími. Některá z vyjádření logických funkcí můžeme vidět níže.

### 5.1.1 Vyjádření pomocí pravdivostní tabulky - ekvivalence

| p | q | p ⇔ q |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

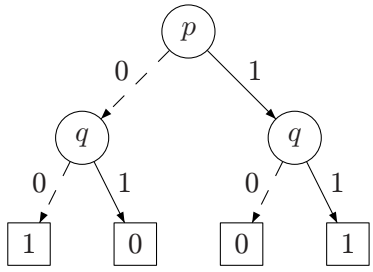
V levé části tabulky jsou všechny kombinace vstupních proměnných v podobě binárních hodnot uspořádány vzestupně. Na pravé straně tabulky jsou zobrazeny hodnoty výstupních proměnných. Pravdivostní tabulky jsou vhodné k vyjádření jakékoliv binární funkce, bez ohledu na počet vstupních proměnných.

### 5.1.2 Vyjádření pomocí KNF - ekvivalence

$$(p \Leftrightarrow p): (\neg p \vee p) \wedge (\neg p \vee p)$$

Konjuktivní normální forma dané formule je formule ekvivalentní s danou formulí a mající tvar konjunkce elementárních disjunkcí. Co se týká KNF a DNF, tak jsou si obě formy zápisu rovnocenné a lze převést jednu formu na druhou použitím Booleovy algebry.

Tyto zmíněné formy však nejsou vhodné pro reprezentaci funkcí, které mají velký počet proměnných. Tyto formy jsou velice náročné na určitá kritéria. Mezi ně můžeme například zmínit náročnost na velikost paměťového místa určeného pro reprezentaci. Jako reakce na tyto problémy vznikly efektivní formy pro manipulaci a reprezentaci velkých logických funkcí v podobě binárních rozhodovacích diagramů (Binary Decision Diagram – BDD). Ekvivalenci jsme si již zobrazili dříve pomocí tabulkového zápisu a konjunktivní normální formy. Nyní si však uvedeme i vyjádření logické funkce ekvivalence pomocí BDD.




Obrázek 9: BDD diagram vytvořený pro ekvivalenci

Teoretické podklady pro BDD vznikaly již na přelomu 50. až 60. let dvacátého století. Jako takové byly binární rozhodovací stromy představeny v roce 1959 C. Y. Lee. Rozmach a obrovský rozvoj v této oblasti byl zaznamenám koncem let 70tých. O tento rozvoj se postarali Boute (1976) a hlavně S. B. Akers (1978). V této době taktéž vzniká název BDD v souvislosti s vyjádřením logických funkcí. O praktické využití BDD se nejvíce postaral R. E. Bryant z univerzity Carnegie Mellon. Tento vědec v polovině 80. let definoval BDD takovým způsobem, jakým se BDD používají prakticky dodnes. Rozšířil doposud dané teorie o použití změny pořadí proměnných - Variable ordering, pro kanonickou reprezentaci Booleovských funkcí a dále o sdílené podgrafy pro možnosti porovnávání funkcí. Využití těchto poznatků přineslo základ pro vytváření efektivních datových struktur, které nazýváme Reduced Ordered Binary Decision Diagrams, a vytváření algoritmů fungujících nad touto strukturou

Všechny funkce jsou reprezentovány jako orientovaný acyklický graf. Tento orientovaný acyklický graf je tvořen třemi typy uzlů: **kořen, neterminální uzly a terminální uzly**.

**Kořen grafu** je pouze jeden a nevedou do něj žádné hrany.

**Vnitřní uzly grafu** jsou označovány jako neterminální uzly, někdy taktéž jako rozhodovací uzly. Jednotlivé uzly odpovídají konkrétní proměnné určité funkce, kterou BDD diagram reprezentuje. Neterminální uzly mají své předky i potomky. To znamená, že do těchto uzlů vedou hrany a vedou taktéž z nich ven. Z jednotlivých vnitřních uzlů **v** vedou

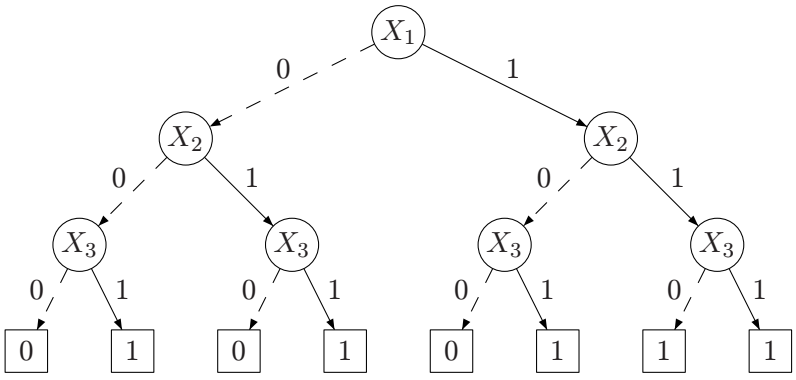
dvě výstupní hrany. Jedna z hran vede do „high potomka“, zkráceně **high(v)**. Hrana *high* odpovídá tomu, že hodnota proměnné uzlu je rovna 1. Druhá z obou hran vede do „low potomka“, zkráceně **low(v)**. Toto označení hrany odpovídá naopak tomu, že hodnota proměnné uzlu je rovna 0.

BDD grafy jsou tvořeny i **koncovými uzly**, které se označují jako listy neboli terminální uzly. Jednotlivé listy, terminální uzly jsou označeny 0 nebo 1 a představují logickou hodnotu, která je konstantní. Do těchto terminálních uzlů vedou hrany pouze dovnitř.

Pro představu je zde obrázek, který představuje BDD graf reprezentující funkci f($X_1$,$X_2$,$X_3$) definovanou pomocí pravdivostní tabulky.

### 5.1.3 Přechodová tabulka:

| $X_1$ | $X_2$ | $X_3$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |




Obrázek 10: BDD diagram vytvořený pomocí tabulky

Každý vnitřní uzel *v* je označen proměnnou *var(v)* a má dvě hrany. Tyto hrany směřují ke dvěma potomkům.

- high(v), jež je označen plnou čarou, toto značení říká, že proměnná je jedničková
- low(v), jež je označen čárkovanou čarou, toto značení říká, že proměnná je nulová

Každý list je dále označen nulou nebo jedničkou. Hodnota funkce je pro dané přiřazení hodnot proměnných určena procházením grafu, které má počátek v kořeni a pokračuje se až k terminálu. Konkrétní rozhodnutí o rozvětvení je provedeno v závislosti na hodnotě dané proměnné v přiřazení. Terminál na konci naší cesty představuje hodnotu funkce. Každá kombinace hodnot vstupních proměnných tedy odpovídá právě jedné cestě z kořenového uzlu stromu až k listu.

#### 5.1.3.1 Získání hodnoty funkce z grafu

V případě, že bychom chtěli z BDD na obrázku č. 10 získat hodnotu pro vstupní proměnné $X_1 = 1$, $X_2 = 1$ a $X_3 = 1$ pak bychom museli postupovat následujícím způsobem:

1. Začneme od kořene našeho stromu na obrázku č. 10. Tomu patří proměnná $X_1$, která je rovna 1. Proto se vydáme po hraně *high*, která je symbolizovaná plnou čarou.
2. V dalším uzlu grafu s proměnnou $X_2$ (v grafu první zprava) se vydáme znovu po hraně *high*, nebot' vstupní hodnota je taktéž rovna 1.
3. V uzlu s proměnnou $X_3$ se vydáme zase po hraně *high*, protože vstupní hodnota je rovna 1.
4. Cesta končí v terminálním uzlu s hodnotou 0. Tato hodnota představuje výstupní hodnotu pro zadané vstupní hodnoty.

Logické funkce tedy vyjadřujeme pomocí grafů. Operace nad Booleovskými funkcemi můžeme tedy implementovat jako grafové algoritmy. Již dříve jsme si uvedli různé tabulkové zápisy jednotlivých logických funkcí. Z těchto tabulkových zápisů by již neměl být problém vytvořit jednoduchý BBD diagram.

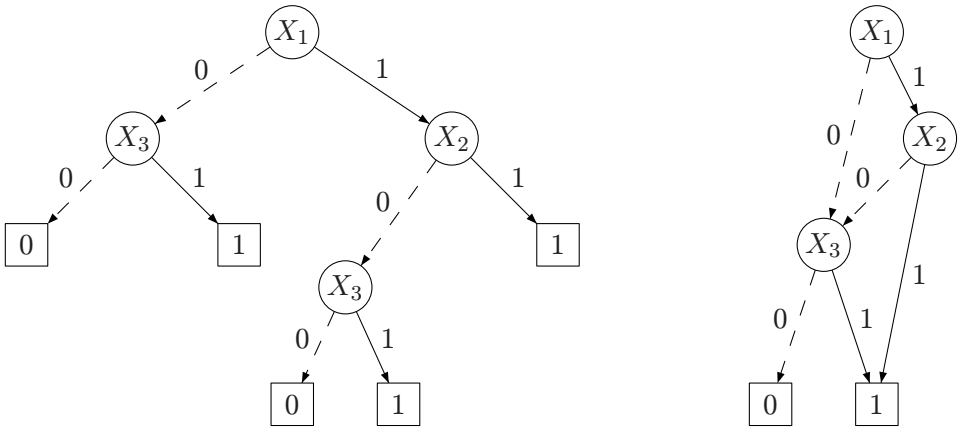
## 5.2 Uspořádané binární rozhodovací diagramy

Binární rozhodovací diagramy (BDD) tvoří abstraktní reprezentaci booleovských funkcí. Při tvorbě však může docházet k situaci, kdy určitou booleovskou funkci můžeme reprezentovat různými BDD. Těchto BDD může existovat větší množství a mohou reprezentovat jednu a tutéž funkci. Z tohoto důvodu se po určité době zavedlo rozšíření v podobě uspořádaných binárních rozhodovacích diagramů, anglicky Ordered Binary Decision Diagrams (OBDDs). Abychom mohli z BDD vytvořit OBDD musíme zajistit určité uspořádání proměnných - Variable ordering. Jedná se o techniku, kterou navrhl R. E. Bryant. Využíváme tedy uspořádání nad množinou proměnných naší funkce. Musí tedy platit určitá nerovnost pro každý uzel $v$ a jeho potomka $w$:

$$\mathbf{var(v) < var(w)}$$

Na obrázku č. 10 je zobrazeno pořadí proměnných v tomto smyslu: x1 $<$ x2 $<$ x3. Toto pořadí však mohlo být zvolené libovolně a původní smysl grafu by to nezměnilo. Požadavek na co nejmenší velikost paměťového prostoru potřebného pro reprezentaci naší funkce je velice důležitý a jedná se o velice složitý problém jak jej zajistit.

OBDD neboli uspořádaný binární rozhodovací diagram je diagram, ve kterém jsou vstupní proměnné seřazeny takovým stylem, že v každé cestě našeho OBDD se jednotlivé proměnné objevují pouze jednou a v tom stejném pořadí. Tato technika řazení má v praxi obrovský vliv na konečnou velikost BDD. Na obrázku č. 11 je uveden příklad zobrazující neuspořádaný (vlevo) a uspořádaný (vpravo) BDD.




Obrázek 11: Vliv uspořádání proměnných na velikost OBDD (Li-C. Wang)

## 5.3 Vznik redukovaných OBDD

Výše popsané OBDD mohou obsahovat různé redundance. Pro jejich zamezení se používá několik pravidel:

1. eliminace nedosažitelných vrcholů
2. odstranění duplikátních listů
3. odstranění duplikátních (vnitřních) vrcholů
4. odstranění redundantních uzlů

**Ad1:** Odstraníme všechny vrcholy, které jsou nedosažitelné z kořene BDD.

**Ad2:** Musíme eliminovat všechny listy (terminály) se stejnou hodnotou. Pokud má tedy OBDD více než jeden list pro 1, zachováme tedy jeden z těchto listů a hrany vedoucí do odstraněných listů přesměrujeme do listu ponechaného.

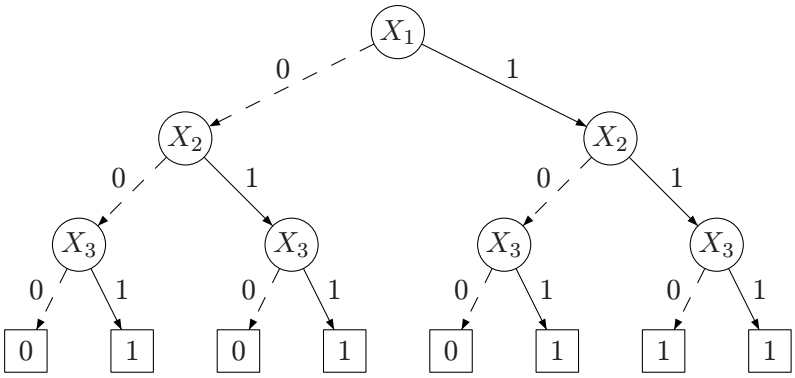
**Ad3:** Když vnitřní vrcholy (neterminální uzly) u a w mají var(v) = var(w), low(v) = low(w) a high(v) = high(w), potom jeden z uzlů v a w odstraníme a všechny hrany vedoucí do odstraněného uzlu převedeme do uzlu druhého. Jednoduše řečeno, pokud z některého uzlu n vedou obě hrany do jednoho uzlu m, odstraníme n a všechny hrany, které do něj vedly, přesměrujeme do m.

**Ad4.** Pokud neterminální uzel v má high(v) = low(v), pak jej odstraníme a všechny vstupní hrany přesměrujeme do low(v).

Pokud tato pravidla aplikujeme na již zmíněné OBDD, tak postupně získáme ROBDD, což je vlastně redukovaný uspořádaný binární rozhodovací diagram. Jedná se o minimalizovaný binární rozhodovací diagram.

### 5.3.1 Vznik ROBDD - příklad

Postup redukce OBDD si popíšeme na obrázku č. 10, který již byl uveden dříve. Pro připomenutí je zobrazen na obrázku č. 12:




Obrázek 12: Binární rozhodovací diagram

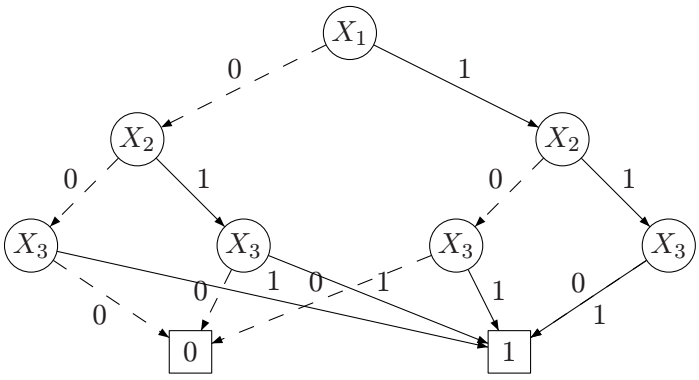
Tento binární rozhodovací diagram (BDD) zachovává uspořádání jednotlivých proměnných $x_1 < x_2 < x_3$. Z tohoto důvodu po aplikování pravidel redukce vznikne redukovaný binární rozhodovací diagram (ROBDD).

V prvním kroku odstraníme z binárního rozhodovacího diagramu všechny nadbytečné terminály a to podle kroku 2 kapitoly 5.3 o vzniku ROBDD. Získáme tak následující diagram na obrázku č. 13:

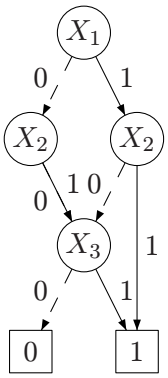
Na obrázku č. 13 tedy vidíme diagram po operaci, která nám odstranila nadbytečné terminály neboli listy. Můžeme si povšimnout, že z původního počtu osmi terminálů jsme dostali jen dva terminály. Nyní budeme aplikovat pravidlo na odstranění opakujících se vnitřních uzlů. Výsledek můžeme vidět na obrázku č. 14.

V posledním kroku aplikujeme na graf z obrázku č. 14 ještě pravidlo na odstranění uzlů, které ze sémantického hlediska nemají žádný význam. Výsledek této operace můžeme vidět na obrázku č. 15.

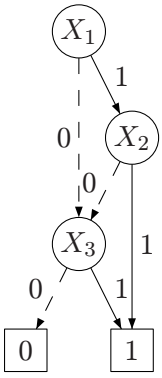
Z patnácti původních uzlů, myšleno terminálů i neterminálů se nám podařilo redukovat binární rozhodovací diagram na redukovaný binární rozhodovací diagram s pěti uzly. Na tomto příkladě názorně vidíme jak velký vliv má redukce na velikost binárních rozhodovacích diagramů.




Obrázek 13: Odstranění nadbytečných terminálů




Obrázek 14: Odstranění opakujících se vnitřních uzlů




Obrázek 15: Odstranění nadbytečných vnitřních uzlů

## 5.4 Vlastnosti ROBDD

Redukované uspořádané BDD mají několik významných vlastností. Největší vlastností asi je, že ROBDD má mnohem méně uzlů než jeho neredukovaná verze. Toto nám rapidně sníží paměťové nároky na reprezentaci takové struktury. Poněvadž nám vypadly některé uzly z grafu, tak i cesta, kterou procházíme od kořene až po terminální symboly je v mnoha případech mnohem menší.

Další vlastností, kterou je třeba zmínit je kanonicita ROBDD.

**Kanonicita ROBDD 1** *Nechť diagram $D_1$ a diagram $D_2$ jsou redukované uspořádané binární diagramy (ROBDD), které vznikly zjednodušením diagramu $D$, potom $D_1 \cong D_2$. $D_1$ a $D_2$ jsou izomorfní.*

Využití kanonicity přináší mnoho užitečných aplikací ROBDD. Je možno například provádět řadu testů nad logickými funkcemi. Mezi tyto testy například patří:

1. testy na splnitelnost logických formulí.
2. testy na tautologii logických formulí.
3. testy zda jsou diagramy ekvivalentní.

V případě, že provádíme testy na splnitelnost logických formulí, funkce, pak musí v grafu existovat minimálně jedna cesta, která vede z kořene do terminálu s hodnotou 1, nebo existuje jediný uzel s hodnotou 0. V případě, že testujeme tautologii logických formulí, funkce a zjistíme, že se jedná skutečně o tautologii, pak má její graf podobu jedničkového terminálu.

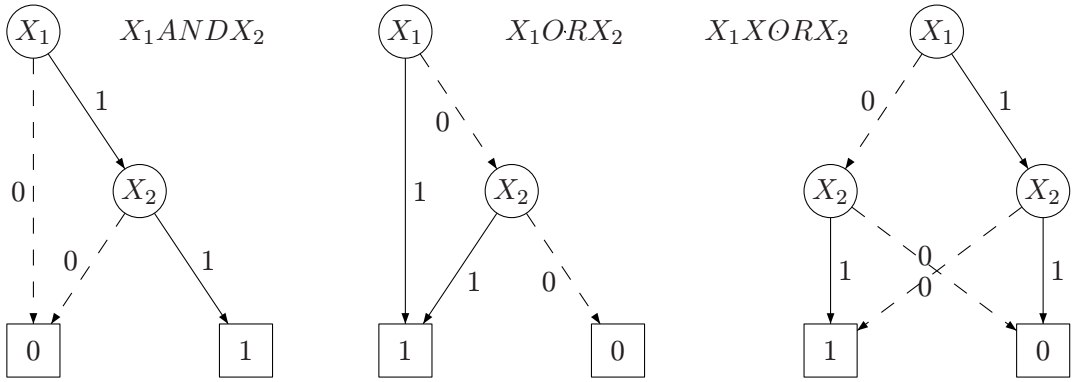
## 5.5 Kanonicita ROBDD

Jak jsme již uvedli, tak nejdůležitější vlastností ROBDD je kanonicita. To znamená, že pokud jsou dvě Booleovské funkce ekvivalentní, pak jejich ROBDD jsou izomorfní. V případě logické ekvivalence se tedy provádí jen kontrola na to, zda jsou dva grafy izomorfní. V případě kontroly tautologií je vliv kanonicity na ROBDD takový, že kontrola tautologie se stává triviální, grafem musí být terminál 1. Uvedeme si ještě obrázky č. 16 a č. 17, které budou znázorňovat ROBDD pro základní logické funkce.

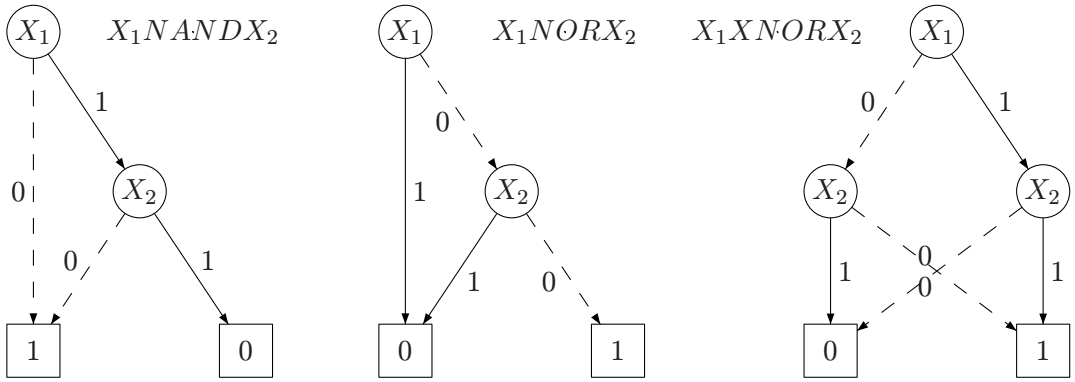
## 5.6 Operace s ROBDD

Jednotlivé ROBDD jsou bez dalšího definování základních operací v podstatě nevyužitelné pro širší použití. Aby bylo možno ROBDD využívat více a následně je třeba spojovat, rozdělovat, či přehazovat a odstraňovat proměnné, bylo potřeba definovat určitá pravidla a hlavně základní operace.

Mezi základní operace patří například: operace *Apply*, operace *Reduce*, operace *Restrict*, operace *Exists* a další.




Obrázek 16: Základní logické funkce AND, OR, XOR vyjádřeny pomocí ROBDD.




Obrázek 17: Základní logické funkce NAND, NOR, XNOR vyjádřeny pomocí ROBDD.

### 5.6.1 Operace Apply

Apply je zaměřena na logické operace. Vstupem této operace jsou dva ROBDD reprezentující logické funkce. Výstupem operace Apply je ROBDD reprezentující výsledek operace. Například *apply(*,*$F_f$*,* $F_g$*)* vrátí výsledek operace logického součinu nad funkcemi $f$ a $g$.

### 5.6.2 Operace Reduce

Tato operace je určená pro redukci OBDD.

### 5.6.3 Operace Restrict

Tato operace provádí dosazení hodnoty za proměnnou. Například *restrict(0, x, $F_g$)* vrátí ROBDD reprezentující funkci $g$ po dosazení 0 za x. Toto se zapisuje jako $f[0/x]$.

### 5.6.4 Operace Exists

Operace Exists odstraňuje závislost funkce na určité proměnné. Například pro operaci *exists(x, g)* je funkce $g[0/x]$ + $g[1/x]$. Další možností je použít zápis *exists($\hat{x}$, g)*. Druhý uvedený zápis provede stejnou operaci pro celou množinu proměnných $\hat{x}$.

### 5.6.5 Negace

Negace přímo nesouvisí s operacemi týkajících se ROBDD, které jsou uvedeny výše. Pokud máme OBDD A, které realizuje funkci $F_x$, tak OBDD B realizující funkci $\neg$ $F_x$ se vytvoří kopií OBDD A a přeznačením listů kopie duální hodnotou.

## 5.7 APPLY operace

Jak již bylo zmíněno dříve, tak operace Apply je zaměřena na logické operace. Vstupem této operace jsou dva ROBDD reprezentující logické funkce a výstupem této operace je jeden ROBDD reprezentující výsledek operace. Můžeme tedy říci, že všechny binární operátory typu Boolean nad ROBDD jsou implementovány pomocí obecného algoritmu APPLY(op, $u_1$, $u_2$) a to tak, že pro dva ROBDDs se vypočte jeden výsledný ROBBD pro výraz typu Boolean, který vypadá takto: $t^{u1}$ *op* $t^{u2}$. Konstrukce operace APPLY, tedy její algoritmus ja založen na Shanonově expanzi, kterou jsme již zmínili v kapitole [2.4]:

$$t = x \rightarrow t[1/x], t[0/x].$$

Všimněme si, že pro všechny Boolean operátory *op* platí následující:
Rovnice č. 1.

$$(x \rightarrow t_1, t_2)\ op\ (x \rightarrow t'_1, t'_2) = x \rightarrow t_1\ op\ t'_1, t_2\ op\ t'_2$$

Shanonova expanze je v algoritmu využita tak, že rozloží obě vstupní funkce (např.: f, g) podle stejné proměnné a následně mezi částmi původní funkce provede požadovanou operaci. Algoritmus, kterým je operace APPLY řešená, vidíme dále. Tento algoritmus prochází vstupní funkce do hloubky a důležité informace si ukládá do dvou hashovacích tabulek. V první tabulce jsou informace, které jsou potřebné pro zefektivnění a zrychlení výpočtu. V druhé hashovací tabulce jsou naopak informace potřebné k vytvoření redukovaného grafu, který je redukován co nejvíce.

```
APPLY[T,H](op, u1, u2)
init(G)

function APP(u1, u2) =
  if G(u1,u2) ≠ empty then return G(u1,u2)
  else if u1 ∈ {0,1} and u2 ∈ {0,1} then u ← op(u1,u2)
  else if var(u1) = var(u2) then
      u ← MK(var(u1), APP(low(u1),low(u2)),APP(high(u1),high(u2)))
  else if var(u1) < var(u2) then
      u ← MK(var(u1), APP(low(u1),u2),APP(high(u1),u2))
  else * var(u1) > var(u2) *
      u ← MK(var(u2), APP(u1,low(u2)),APP(u1,high(u2)))
  G(u1,u2) ← u
  return u
end APP

return APP(u1,u2)
```

V případě, že začínáme z kořene obou ROBDD, můžeme sestrojit výsledný ROBDD rekurzivní konstrukcí *low* a *high* větve a následně vytvoříme nový kořen z původních dvou ROBDD. Opět musí být zajištěno, že výsledek je redukovaný. Vytváříme uzly pomocí volání metody *MK*, která je uvedená níže. Tímto se vyhneme exponenciálnímu nárůstu rekurzivních volání. Je tedy použito dynamické programování. Algoritmus, který toto vše provádí, je zobrazen výše.

Dynamické programování je implementováno použitím tabulky výsledků *G*. Každý vstup *(i,j)* je prázdný, nebo obsahuje dříve vypočtený výsledek funkce *APP(i, j)*. Algoritmus rozlišuje mezi čtyřmi rozdílnými případy. První z nich představuje situaci, kde oba argumenty jsou terminální uzly. Zbývající tři představují situace, kdy nejméně jeden argument je proměnná typu uzel.

Jestliže jsou $u_1$ a $u_2$ terminály, pak nový terminální uzel má hodnotu získanou použitím operátoru *op* aplikovanou na dvě pravdivostní hodnoty. (Připomeňme, že terminální uzel 0 je reprezentován jako uzel s identitou 0, to stejné platí i pro 1.)

Jestliže je nejméně jeden z $u_1$ nebo $u_2$ neterminál, pak musíme postupovat v souladu s indexem proměnné. Jestliže uzly mají stejný index, dvě *low* větve jsou v páru, pak z nich *APP* vypočteme rekurzivně. Stejným způsobem to provedeme i pro *high* větev. To odpovídá rovnici č. 1. Jestliže mají rozdílné indexy, pak postupujeme porovnáním uzlu s nejnižším indexem *low* a *high* větve, které následují. Toto odpovídá následující rovnici č. 2., která platí pro všechny $t$.

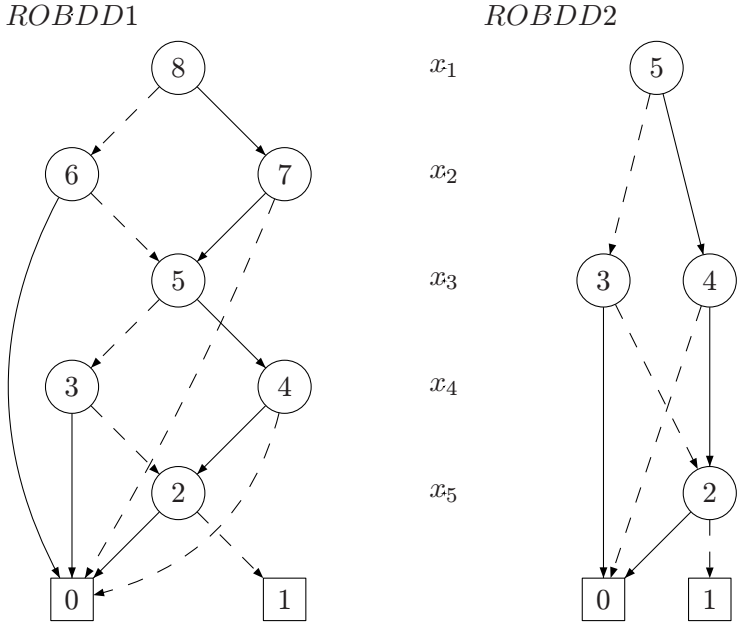
Rovnice č. 2.

$$(x_i \rightarrow t_1, t_2)\ op\ t = x_i \rightarrow t_1\ op\ t, t_2\ op\ t$$

Jelikož uvažujeme index z terminálů, pak nastane situace taková, že jeden index z terminálů bude větší než index z neterminálů. U zbývajících dvou případů $var(u_1) < var(u_2)$ a $var(u_1) > var(u_2)$, budeme uvažovat situaci takovou, že jeden z uzlů je terminál.

Obrázek č. 18 zobrazuje dva ROBDD, na které bude aplikován algoritmus operace *Apply* pro výpočet konjunkce. Obrázek č. 19 zobrazuje řešení použitého algoritmu. Na obrázku č. 20 je zobrazen strom z argumentů rekurzivně volaných *APP*. Čárkované uzly v tomto stromu znázorňují, že hodnota uzlu byla dříve vypočtená a není přepočítávána přímo použitím dynamického programování. Na obrázcích si všimněme, jak jsou jednotlivé páry uzlů z dvou ROBDD kombinovány a vypočítávány.

K analýze složitosti operace Apply necháme $|u|$ označené číslem z uzlů, kterého můžeme dosáhnout z *u* v ROBDD. Předpokládáme, že *G* může být implementováno v konstantním čase a to pro vyhledávání *(lookup)*, i pro vložení *(insert)*. Užitím dynamického programování je generováno nejvíce $|u_1||u_2|$ volání operace Apply. Každé volání je uskutečněno v konstantním čase. Celkový čas je tedy $O(|u_1||u_2|)$.



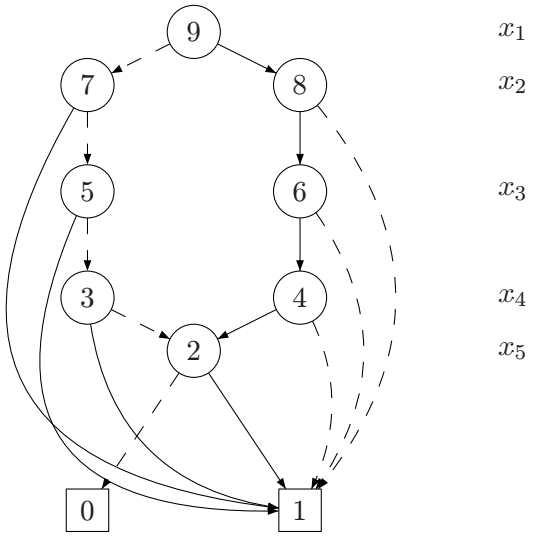

Obrázek 18: ROBDD1, ROBDD2 na které je aplikována Apply pro výpočet konjunkce.

### 5.7.1 Konstrukce ROBDD

Při tvorbě ROBDD vyvstává otázka, jak provést konstrukci a redukování OBDDD. Existují dva možné způsoby. Prvním z nich je, že nejprve vytvoříme OBDD a poté provedeme redukci tohoto OBDD a vytvoříme tak ROBDD. Druhým způsobem, který je více žádoucí je, že provádíme redukování během konstrukce ROBDD.

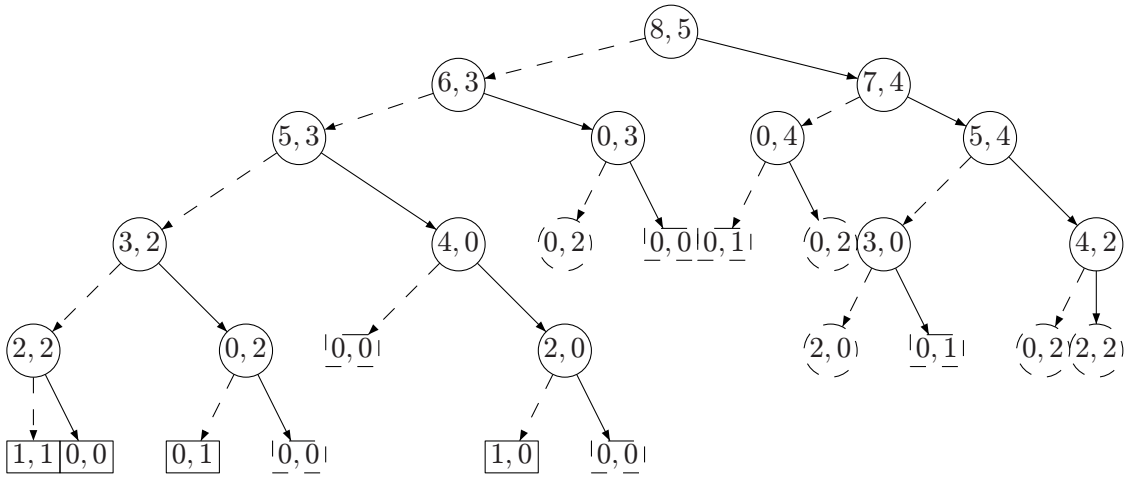
K popisu, jak lze toto udělat, potřebujeme explicitní reprezentaci ROBDD. Uzly budeme reprezentovat jako čísla 0, 1, 2, 3, ... . Čísla 0 a 1 budeme mít rezervovány pro

$ROBDD1 \wedge ROBDD2 =$




Obrázek 19: Výsledek Apply operace použité na dva ROBDD.

terminální uzly. Uspořádané proměnné $X_1 < X_2 < X_3 < ...X_n$ budeme reprezentovat jejich indexy 1, 2, 3, ..., n. ROBDD budeme ukládat do tabulky $T : u \mapsto (i, l, h)$, kde budeme mapovat uzel *u* na jeho tři atributy *var(u) = i, low(u) = l a high(u) = h*.




Obrázek 20: Apply operace užitá na dva ROBDD

**5.7.1.1 Funkce MK** Uspořádání nám zajistí, že OBDD budou konstruovány již redukované. Toto je velice nutné k určení, zda-li existuje v trojici *(i, l, h)* uzel *u* s *var(u) =*

*i, low(u) = l a high(u) = h*. Předpokládáme tedy přítomnost z tabulky $H : (i, l, h) \mapsto u$ mapující trojici *(i, l, h)* z indexů proměnných *i*, uzlů *l, h* do uzlu $u$. Tabulka $H$ je inverzní k tabulce *T*, tj. pro proměnné uzlu *u*, *T(u) = (i, l, h)* a to tehdy a jen tehdy pokud *H(i, l, h) = u*. Operace potřebné pro tyto dvě tabulky jsou:

| | | |
|---|---|---|
| T: | $u \mapsto (i, l, h)$ | |
| | init(T) | - inicializace T obsahující pouze 0 a 1 |
| | $u \leftarrow add(T, i, l, h)$ | - alokování nového uzlu $u$ s atributy *(i, l, h)* |
| | var(u), low(u), high(u) | - vyhledání atributu z $u$ v $T$ |
| H: | $(i, l, h) \mapsto u$ | |
| | init(H) | - inicializace prázdné $H$ |
| | $b \leftarrow member(H, i, l, h)$ | - kontrola zda *(i, l, h)* je v $H$ |
| | $u \leftarrow lookup(H, i, l, h)$ | - nalezení *H(i, l, h)* |
| | insert(H, i, l, h, u) | - provedení *(i, l, h)* mapování $u$ do $H$ |

Algoritmus funkce *MK* můžeme vidět dále.

```
MK[T, H](i, l, h)
  if l = h then return l
  else if member(H, i, l, h) then
  return lookup(H, i, l, h)
  else u ← add(T, i, l, h)
  insert(H, i, l, h, u)
  return u
```

Budeme předpokládat, že všechny tyto operace mohou být proveditelné v konstantním čase *O(1)*.

Funkce *MK[T, H](i, l, h)* hledá tabulku *H* pro uzel s proměnnou označenou indexem *i* a *low* a *high* větev *l, h* a vrací odpovídající uzel v případě, že existuje. Pokud ne, tak je vytvořen nový uzel *u*. Následně je vložen do tabulky *H* a je vrácena jeho identita. Složitost *MK* je předpokládána pro základní operace z *T* a *H O(1)*. OBDD bude redukované, jestliže uzly jsou vytvářeny pouze skrze užití *MK*. Ve jméně funkce *MK* a jejím algoritmu je uvedeno značení *[T, H]*, které označuje, že *MK* je závislá na globální datové struktuře *T* a *H*. Vynecháváme však argumenty v případě, že je vyvoláme jako část z algoritmu.

## 5.8 Reprezentace BDDs

Pro reprezentaci BDDs bylo vymyšleno několik způsobů, jak implementovat konkrétní BDD, jež byly popsány výše. Jednotlivé způsoby se ve svém principu výrazně liší. Některé ze způsobů implementace nabízejí několik výhod, zatímco jiné jsou pro implementaci výhodné méně. Velice známým způsobem je reprezentace BDD pomocí binárního stromu, nebo způsob reprezentace BDD pomocí knihovny CUDD.

V době kdy jsem studoval materiály týkající se BDDs jsem nalezl několik balíčků, neboli knihoven, které se zabývají manipulací a reprezentací BDD. Nebudu je zde však popisovat a detailně rozebírat. Chtěl bych však uvést jejich pojmenování a krátký popis, aby si případný čtenář, kterého tato kapitola zaujala, mohl vyzkoušet různé způsoby reprezentace BDDs pomocí dalších balíčků a knihoven.

**CUDD** Balíček CUDD [18] přináší velké množství funkcí pro práci s binárními rozhodovacími diagramy (BDD), algebraickými rozhodovacími diagramy (ADD), binárními rozhodovacími diagramy s potlačenou nulou (ZDD) a mnoha dalšími.

**BuDDy** Další balíček určený pro práci s BDD. Balíček BuDDy [19] byl vytvořen jako součást doktorantské práce J. Lind-Nielsenem na univerzitě v Kodani. Jeho poslední, zatím uvedená verze oproti první verzi, obsahuje všechny důležité operace, které jsou potřeba pro práci s BDD a mnoho dalších vylepšení.

**ABCD** Balíček ABCD [20] je distribuován v experimentální verzi a je určen pro manipulaci s BDD. V tomto balíčku je rozdílně řešen Garbage Collector, dále je zde použita jiná integrace unikátních hashovacích tabulek uzlů. Ta se provádí s využitím otevřeného adresování. Toto umožnilo snížit velikost struktury uzlu až na polovinu.

**CAL** Tato knihovna [21] je výsledkem práce autorů se jmény R. Ranjan a J. Sanghavi. Knihovna CAL je volně distribuovaná. Slouží pro práci s BDDs a je založená na algoritmu prohledávání do šířky. Zde, v této knihovně je Garbage Collector řešen taktéž jinak a to pomocí počítání referencí.

**CMU BDD, CrocoPat, JavaBDD, TUD BDD** Další balíčky, se kterými je možno se setkat.

## 5.9 Knihovna CUDD

Informace pro vypracování této části diplomové práce jsem čerpal z *CUDD: CU Decision Diagram Package* [18].

CUDD package, neboli Colorado University Decision Diagram package je balík, který byl vyvíjen Fabiem Somenzi a jeho spolupracovníky na univerzitě, jejíž název je uveden v názvu balíku CUDD. S vývojem tohoto balíku bylo započato roku 1996. Poslední verzi balíku s označením 2.4.1 je možno zhlédnout na internetových stránkách tohoto projektu zde: [18]. Jedná se o velice propracovaný balík s mnoha funkcemi. Jen v základním balíku CUDD je k dispozici více než čtyři sta funkcí. Tento balík podporuje operace s BDD (Binary Decision Diagram), ADD (Algebraic Decision Diagram), a v neposlední řadě i s

VIS (Veryfication Interacting with Synthesis). Projekt CUDD je napsán v jazyce C. Je však možno nalézt i rozhraní napsané v C++.

Tento balík je velice rozsáhlý a komplexní. Jsou zde použity různé techniky a způsoby práce s BDD. Chtěl bych zde uvést některé z vlastností balíku CUDD, například: acyklické grafy jsou zde reprezentovány pomocí tabulek. Konkrétním uzlům grafu odpovídají jednotlivé řádky v dané tabulce. Veškeré operace týkající se BDD jsou řešeny pomocí operací nad zmíněnou tabulkou. V balíku CUDD je možno využívat komplementární data. CUDD používá jak normální, tak i negované hrany. Velice významnou vlastností balíku CUDD je taktéž to, že obsahuje velké množství algoritmů, které vylepšují uspořádání proměnných. Toto byl jen minimální výčet vlastností, které CUDD umožňuje, avšak dalších vlastností je mnohem více.

Balíček CUDD lze používat několika způsoby:

- **Black box**, neboli „černá skříňka“. V případě používání balíčku CUDD jako černé skříňky, program využívá jen funkce, které jsou zpřístupněné, případně takové funkce, které jsou viditelné jen z venku této černé skříňky.

- **Clear box**, neboli „průhledná skříňka“. Jako clear box se balík CUDD využívá zejména při psaní sofistikovanějších aplikací. V tomto režimu je možno vytvářet své vlastní funkce a následně je přidávat do balíčku CUDD.

- Prostřednictvím **rozhraní**. Pro C++ je již rozhraní obsaženo v distribuci CUDD. Je možno díky tomuto objektově orientovanému jazyku osvobodit programátora od správy paměti. Jazyk C++ je navíc doporučován, neboť využívá také rychlé prototypování. Tím je ulehčeno v mnoha případech hodně práce. Co se týká jazyka PERL5 a jeho rozhraní, tak ten má taky stejnou vlastnost jako jazyk C++ týkající se správy paměti. Jeho rozhraní je však distribuované samostatně.

### 5.9.1 Práce s datovými strukturami

Práce s datovými strukturami jako je například ROBDD se skládá z velkého množství jednoduchých operací, které jsou nad danou strukturou vykonávány. Tato struktura, konkrétně ROBDD, jak již bylo zmíněno dříve, může při praktickém použití dosahovat obrovské velikosti a může obsahovat nezměrné množství uzlů. Proto je z důvodu lepšího využití paměti potřebné zvolit si správnou reprezentaci uzlu a vědět, jak s těmito uzly správně pracovat (referencovat, dereferencovat, ...). V knihovně CUDD se k reprezentaci uzlu používá struktura *DdNode*. K práci s jednotlivými uzly se v této knihovně využívá *DdManager*. Struturu uzlu *DdNode* si popíšeme. Nejprve se však podívejme, jak lze jednoduše vytvořit ROBDD pomocí knihovny CUDD.

### 5.9.2 Vytvoření jednoduchého ROBDD

Po důkladném prostudování dokumentace, která je přístupná na internetových stránkách popisujících knihovnu CUDD [18], je možno s touto knihovnou velice snadno pracovat.

Pokud by si chtěl někdo vyzkoušet jednotlivé funkce, které tato knihovna poskytuje, doporučuji mu navštívit již zmíněné internetové stránky. Zde se nebudu zabývat každou funkcí, každou vlastností, kterou knihovna CUDD má. Ukážeme si krátký příklad kódu, na kterém uvidíme, jak se vytváří jednoduchý ROBDD:

```
#include <stdio.h>
#include ”include/cudd.h”
#include ”include/util.h”
int main() {
      DdManager *manager;
      DdNode *f1,*f2,*f,*tmp,*var;
      int i;
      double k;
      manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS,
                  CUDD_CACHE_SLOTS, 0);
      //inicializace manažeru - DdManager
      f1=Cudd_ReadOne(manager); //vytvoření bdd x0 * x1 * x2
      Cudd_Ref(f1);
      for(i = 2; i >= 0; i--){ // cyklus přidávání uzlů do BDD
            var = Cudd_bddIthVar(manager, i);
            tmp = Cudd_bddAnd(manager, var, f1);
            Cudd_Ref(tmp);
            Cudd_RecursiveDeref(manager, f1);
            f1 = tmp;
      }
      f2 = Cudd_ReadOne(manager); //vytvoření bdd x0’*x1’*x2’
      Cudd_Ref(f2);
      for(i=2; i >= 0; i--){ // cyklus přidávání uzlů do BDD
            var = Cudd_bddIthVar(manager, i);
            tmp = Cudd_bddAnd(manager, Cudd_Not(var),f2);
            Cudd_Ref(tmp);
            Cudd_RecursiveDeref(manager, f2);
            f2 = tmp;
      }
      f = Cudd_bddOr(manager, f1, f2); // vytvoření celého BDD
      Cudd_Ref(f);
      Cudd_RecursiveDeref(manager, f1);
      Cudd_RecursiveDeref(manager, f2);
      Cudd_Quit(manager);
      return 0;
}
```

Předchozí zdrojový kód je určen pro vytvoření ROBDD pro funkci:
f = $(x_0 \wedge x_1 \wedge x_2) \vee (\neg x_0 \wedge \neg x_1 \wedge \neg x_2 s)$.

Na začátku programu je inicializován *DdManager* voláním funkce *Cudd_init()* s příslušnými parametry. Tento *DdManager* se stará o správu paměti. Po inicializaci *DdManageru* přecházím k vytváření ROBDD. Výsledný ROBDD je v tomto případě vytvářen tak, že se ve dvou cyklech „for" vytvoří ROBDD pro každý term. Tyto termy jsou pak spojeny do jednoho výsledného diagramu.

ROBDD se vytváří zdola nahoru. To znamená, že se nejprve pomocí funkce *Cudd_ReadOne* vytvoří konstantní uzel *1* a následně se v cyklu přidávají další uzly. V případě, že daný uzel již existuje, tak je vrácen ukazatel na tento existující uzel. Přidávání dalších uzlů je řešeno tak, že nejprve vytvoříme uzel, jedná se v podstatě o proměnnou, v manažeru pomocí funkce *Cudd_bddIthVar*. Tuto proměnnou následně přidáme pomocí funkce *Cudd_bddAnd* do ROBDD. K negaci proměnné se využívá funkce *Cudd_Not*, které je jako parametr předáváno jméno proměnné, kterou chceme znegovat. Mezivýsledky, které postupně vznikají, se ukládají do pomocné proměnné, která je pojmenována *tmp*. Tyto mezivýsledky musí být při vzniku vždy referencovány. To se děje vlivem funkce *Cudd_Ref*. Při zániku musí být mezivýsledky naopak dereferencovány. To se provádí funkcí *Cudd_RecursiveDeref*. Dobře provedené referencování a dereferencování je důležité pro správnou funkčnost garbage collectoru. Garbage collector řeší uvolňování paměti. Provádí se to takovým stylem, že odstraňuje data, která již nejsou potřeba. Špatná práce s referencováním a dereferencováním jednotlivých uzlů má za následek plýtvání s pamětí. V případě, kdy se řeší větší úlohy a projekty, může vést k rychlému zaplnění paměti. V závěru jsou oba ROBDD spojeny do jednoho výsledného ROBDD užitím funkce *Cudd_bddOr()*. Na konci programu vidíme také funkci *Cudd_Quit(manager)*, pomocí které ukončíme *DdManager*.

### 5.9.3 Struktura DdNode

Při používání knihovny CUDD a při čtení textů, které se touto knihovnou zabývají, zjistíme, že rozhodovací diagramy a jejich uzly jsou realizovány strukturou nazvanou DdNode, která má následující zápis.

```
struct DdNode {
   DdHalfWord index;
   DdHalfWord ref;
   DdNode *next;
   union {
     CUDD_VALUE_TYPE value;
     DdChildren kids;
   } type;
};
```

Konkrétní význam jednotlivých proměnných můžeme vidět níže.

***Index*** Index obsahuje jméno proměnné, které představuje zároveň pojmenování uzlu. Jedná se o neměnný atribut, který ukazuje pořadí vytvoření dané proměnné. První proměnná je označena indexem 0, poslední vytvořená proměnná je označená nejvyšším indexem.

***Ref*** Ref uchovává informaci o počtu referencí na konkrétní uzel. Tato informace je využívána garbage collectorem, který uzly, jež mají ref count nulový, postupně odstraní.

***Next*** Next je proměnná typu DdNode. Tato proměnná je ukazatelem na další uzel v hashovací tabulce.

***Union*** Union v případě, že se jedná o neterminální uzel, obsahuje ukazatele na potomky daného uzlu. Pokud se jedná o terminál, tak obsahuje hodnotu tohoto terminálu.

### 5.9.4 DdManager

Jednotlivé uzly Binary Decision diagramů jsou uloženy v hashovacích tabulkách. Tyto tabulky jsou pojmenovány *unique tables*. Hlavním účelem těchto tabulek je zaručit, že každý uzel, který je v ní uložený je jedinečný. Touto jedinečností je myšleno to, že v tabulce již není žádný další uzel, který má stejné pojmenování stejné proměnné a se stejnými potomky. Díky tomu, že je zaručena jedinečnost každého uzlu, jsou všechny rozhodovací diagramy kanonické. DdManager tedy obsahuje hashovací tabulku a několik proměnných, které je možno využívat pro práci s rozhodovacími diagramy. Před začátkem práce musí být DdManager inicializován pomocí funkce *Cudd_Init* s příslušnými parametry této funkce. Po ukončení práce musíme DdManager zrušit. To se provádí pomocí funkce *Cudd_Quit*. Tato funkce musí obsahovat taktéž příslušný parametr a tím je samotné jméno konkrétního manageru.

Knihovna CUDD obsahuje jen několik globálních proměnných a několik statistických proměnných. Většina proměnných je totiž udržována v manageru. Je tedy možné mít v jedné aplikaci několik aktivních managerů, přičemž ukazatel na konkrétní manager, rozhoduje nad jakou skupinou dat bude funkce pracovat.

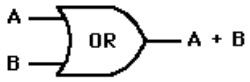
## 5.10 Použití BDDs

Binary Decsion Diagrams jsou v praxi většinou nasazovány v oblastech jako je návrh, verifikace, testování a optimalizace digitálních číslicových obvodů. V poslední době se však praktické využití BDD stále více rozmáhá a rozšiřuje i do jiných oblastí. S BDD je možné se setkat při řešení různých problémů například i v matematické logice. BDD se objevuje také v oblasti umělé inteligence, kde vědci vymysleli pravdivostní systém založený na ROBDD.

### 5.10.1 Příklad použití BDDs

V této části si uvedeme příklad, na kterém si ukážeme jak z jednoduchého logického obvodu vytvořit OBDD.

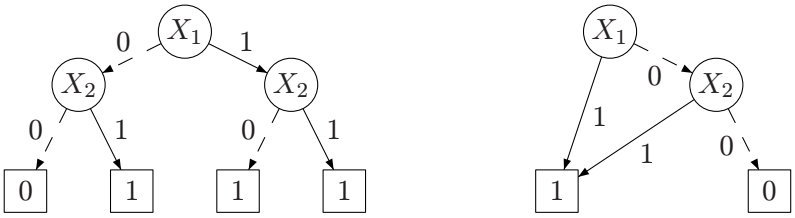
Pro začátek bych chtěl uvést, jak vypadá BDD pro logické členy OR a AND. Následně si ukážeme konkrétní zapojení těchto dvou členů v obvodu a k tomuto zapojení vytvoříme BDD diagram. Na brázku č. 21 lze vidět logický člen OR, tedy jeho blokové schéma. Tabulku a BDD diagram pro tento logický člen můžeme spatřit dále. BDD diagram je na obrázku číslo 22.




Obrázek 21: Logický člen OR

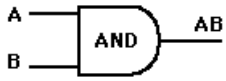
#### 5.10.1.1 OR - tabulkový zápis

| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |




Obrázek 22: BDD, ROBDD diagram vytvořený pro OR

Zobrazili jsme si tedy logický člen OR. Nyní si ukážeme i logický člen AND. Na obrázku č. 23 vidíme blokové schéma logického členu AND.

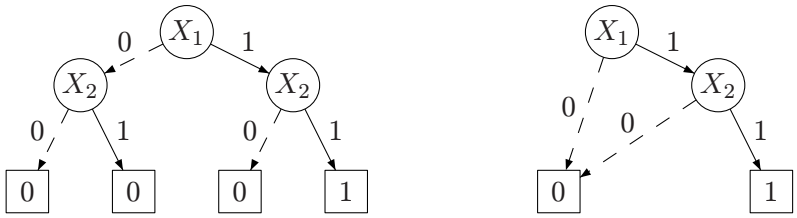



Obrázek 23: Logický člen AND

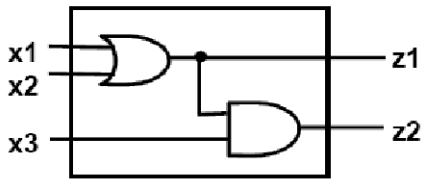
Pro tento člen lze vytvořit také BDD diagram. Ten můžeme vidět na obrázku č. 24. Na tomto obrázku jsou zobrazeny dva BDD diagramy, jeden je již upraven do zjednodušené formy, která se používá pro řešení úloh v různých logických obvodech.

V této chvíli máme určitou představu, jak vypadají jednotlivé BDD diagramy pro konkrétní logické členy. Těchto členů je mnohem více, pro náš příklad nám však postačí tyto dva členy. Příklad neboli obvod, ke kterému směřujeme lze vidět na obrázku č. 25.

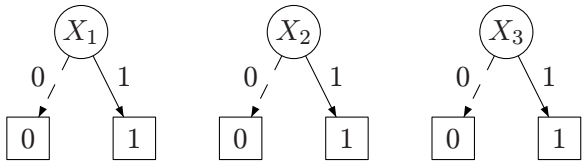
Zkusíme nyní pro tento obrázek vytvořit BDD diagram. Na obrázku si můžeme všimnout topologicky setříděných vstupů a výstupů, které jsou setříděny v tomto pořadí: *X1, X2, X3, Z1, Z2*. Jednotlivé vstupy, neboli proměnné jsou setříděny v následující pořadí:




Obrázek 24: BDD, ROBDD diagram vytvořený pro AND



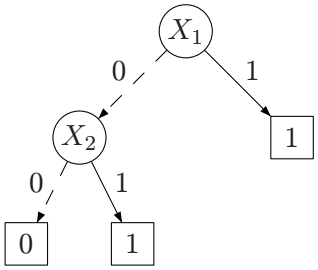

Obrázek 25: Obvod tvořený členy OR, AND




Obrázek 26: OBDD(X1), OBDD(X2), OBDD(X3)

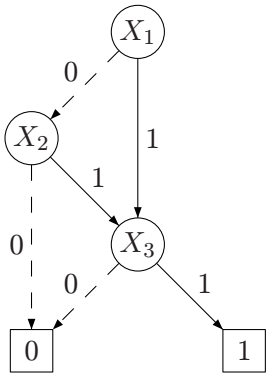
*X1* → *X2* → *X3*. Na obrázku č. 26 vidíme OBDD diagramy pro jednotlivé proměnné X1, X2, X3.

Z obrázku víme, že X1 a X2 vstupují do logického členu OR. Provedeme nad nimi operaci Apply. Vznikne nám OBDD, které můžeme vidět na obrázku č. 27. *OBDD(Z1) = OBDD(X1) + OBDD(X2)*.




Obrázek 27: OBDD pro Z1

Na posledním obrázku můžeme vidět již výsledný OBDD pro Z2. Tedy OBDD(Z2) = OBDD(X3) . OBDD(Z1).




Obrázek 28: OBDD pro Z2

Ukázali jsme si tedy jak převést určitý obvod z praktické oblasti na BDD diagram. Taky jsme se již dříve zmínili o verifikačním nástroji SMV (Symbolic model verifer), který pro verifikování využívá právě probírané BDD. Pomocí tohoto nástroje můžeme tedy snadno verifikovat hardwarové obvody, které jsou tvořené logickými členy. V kapitole č. 7 si pak ukážeme jak jednotlivé obvody přepsat do jazyka SMV a jak v tomto nástroji provádět verifikaci konkrétních vlastností obvodů a dalších systémů.

# 6 Nástroje založené na BDDs

V dnešní době existuje několik nástrojů, které jou založeny na BDDs, neboli binárních rozhodovacích diagramech. Mezi nejznámější patří Symbolic model verifer - SMV, Cadence SMV a NuSMV.

## 6.1 SMV

SMV je nástroj pro symbolické ověřování modelů. Jedná se o nejpoužívanější model-checker pro verifikaci modelů hardwaru. Obsahuje vlastní jazyk - SMV language. Z tohoto nástroje vzniklo několik různých větví:

- CMU SMV: což je původní SMV
- Cadence SMV: Cadence Berkeley Lab, byl základem pro komerční nástroj Incisive Formal Analysis
- NuSMV: aktuální verze 2.4.3

SMV nástroj je možné stáhnout ze stránek School of computer science [25]. Konkrétní adresa, kde je možné stáhnout nástroj SMV je zde [26]. Na uvedeném odkazu je možné nalézt instalační soubory pro Linux, zdrojové kódy, předkompilované binární soubory a manuál [27] zabývající se SMV.

Instalační instrukce nástroje SMV pro Unix, resp. pro Windows jsou uvedeny na [30], resp [31]. Instrukcí uvedených na tomto odkazu jsem využil při instalaci nástroje SMV v rámci této diplomové práce. SMV byl nainstalován na operačním systému Ubuntu 6.06 LTS - Dapper Drake. Pro práci a zápis modelů v jazyce SMV byl zvolen textový editor, který je součástí Ubuntu. Spouštění a zadávání příkazů bylo prováděno pomocí Terminálu.

Více informací, které se zabývají již praktickými záležitostmi týkajících se nástroje SMV, modelů různých systémů, verifikováním modelů a dalšími záležitostmi, je uvedeno v následující kapitole pojmenované *Příkladová studie*.

## 6.2 Cadence SMV

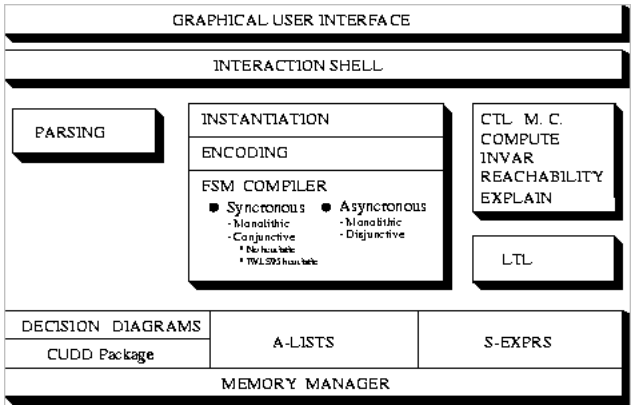
Nástroj pro symbolické ověřování modelů založený na SMV. Má lepší vyjadřovací schopnosti popisu jazyka. Umožňuje několik forem specifikace, včetně CTL a LTL, konečných automatů a dalších.

## 6.3 NuSMV

Jako dva předešlé, tak i tento nástroj je určen pro symbolické ověřování modelů. Nástroj NuSMV je možno stáhnout ze stránek uvedených zde [32]. Na těchto stránkách můžeme taktéž nalézt manuál a tutoriál NuSMV. NuSMV vznikl zefektivněním, novou implementací a rozšířením nástroje SMV. Zefektivnění bylo provedeno i v rámci uživatelského

rozhraní, bylo vytvořeno grafické uživatelské rozhraní - gNuSMV. Nástroj je tedy možné využívat interaktivně pomocí příkazové řádky, nebo pomocí grafického uživatelského rozhraní. Abychom mohli využívat grafické uživatelské rozhraní, musíme mít nainstalovány různé balíky a nastaveny správné cesty. Návod, pomocí kterého lze snadno vše připravit pro používání grafického uživatelského rozhraní NuSMV je na těchto stránkách [33]. V rámci této práce jsem vyzkoušel možnosti nástroje NuSMV v režimu práce na příkazové řádce, tak i v jeho grafickém uživatelském rozhraní. Tak jako nástroj SMV, tak i NuSMV jsem nainstaloval na operačním systému Ubuntu 6.06 LTS - Dapper Drake. Pro práci a zápis modelů byl tak jako v předešlém případě zvolen textový editor, který je součástí Ubuntu. Spouštění a zadávání příkazů bylo prováděno pomocí Terminálu, případně přímo v grafickém rozhraní gNuSMV.

NuSMV je tedy symbolický model checker, který kombinuje klasické techniky založené na BDD s technikami založenými na SAT (splnitelnost výroku). Distribuování tohoto nástroje je od verze číslo 2 s OpenSource licencí. Uživatelům je tedy nabídnuta možnost zapojení se do vývoje a zdokonalování tohoto nástroje. Ačkoliv je možno provádět různé úpravy, tak architektura tohoto nástroje je přesně daná. Komponenty a jednotlivé funkce tohoto nástroje jsou izolovány a rozděleny do modulů. Mezi těmito moduly je definováno rozhraní. Díky tomuto rozhraní je minimalizována jakákoliv potřeba provádět změny jádra nástroje NuSMV. Blokové schéma architektury NuSMV je vidět na obrázku č. 29. Tento obrázek je přejat z dokumentu *NUSMV: a new Symbolic Model Verifier* [34], který popisuje nástroj NuSMV.




Obrázek 29: Architektura systému NuSMV

Zmíněné jádro systému poskytuje nejnižší úroveň funkcionality. Jedná se například o dynamickou alokaci paměti nebo manipulaci se základními datovými strukturami. Jádro poskytuje základní BDD funkce, které jsou převzaty přímo z knihovny CUDD BDD. Další poznatky týkajících se NuSMV je možno najít v dříve uvedeném dokumentu *NUSMV: a new Symbolic Model Verifier* [34] a také v diplomové práci Michala Výmoly [11], ze které jsem taktéž čerpal některé informace. V této diplomové práci se zabývá porovnáním softwarových nástrojů (NuSMV, DiVinE) určených pro model checking.

V rámci této práce jsem prováděl verifikaci modelů, které budou uvedeny později, také v nástroji NuSMV. Verifikace modelů byla prováděna pomocí příkazové řádky (interactive shell), i pomocí grafického uživatelského rozhraní (gNuSMV).

### 6.3.1 Práce s příkazovou řádkou NuSMV

Při práci s nástrojem NuSMV pomocí příkazové řádky je potřeba nejprve NuSMV spustit. To se provádí pomocí příkazu: *NuSMV -int*. Aby se nám NuSMV na příkazové řádce podařilo spustit, tak se musíme v adresářové struktuře nacházet na místě, kde jsme nástroj NuSMV předem nainstalovali. Po zadání uvedeného příkazu se nám spustí NuSMV a zobrazí se nám několik informací o verzi tohoto nástroje, informace o domovské internetové stránce a emailový kontakt. V této chvíli můžeme spouštět různé výpočetní kroky zadáváním systémových příkazů. Pro uživatele, kteří by si nevěděli rady, je zpracována podrobná nápověda. Zobrazení všech příkazů lze provést pomocí *help* příkazu. Uvedením příkazu *help*, případně uvedením *-h* za konkrétní příkaz, zobrazíme nápovědu a bližší informace k tomuto příkazu. Ukázku použití příkazu *help* můžeme vidět zde:

```
NuSMV > read_model help
usage: read_model [-h] [-i <file>]
      -h       Prints the command usage.
      -i <file>Reads the model from the specified <file>.
```

V případě, že bychom chtěli provést simulaci, či verifikaci modelu, který je již dříve vytvořen (*krizovatka.smv*), tak zapíšeme několik následujících příkazů.

```
NuSMV > read_model -i krizovatka.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
```

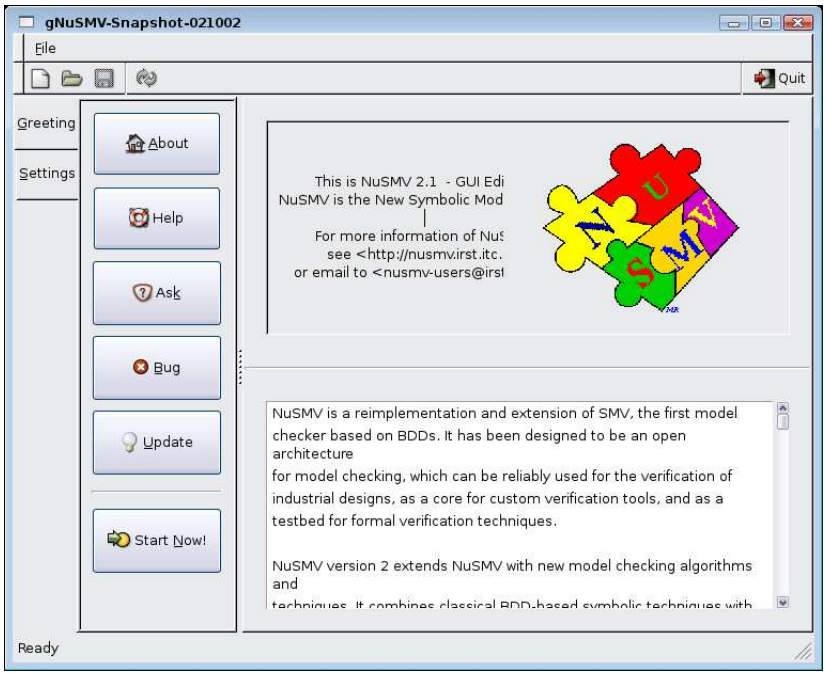
Tyto čtyři příkazy je potřeba zapsat z toho důvodu, aby byl NuSMV schopen vykonat simulaci, či verifikaci. V této chvíli můžeme provést verifikaci pomocí příkazu *check_spec*. Pokud bychom chtěli verifikovat další CTL formule, tak použijeme příkaz *check_spec -p „CTL formule“*. NuSMV nám následně poskytne odpověď, zda je konkrétní formule pravdivá či nikoliv. Pokud bychom chtěli ukončit práci s nástrojem NuSMV, tak to provedeme pomocí příkazu *quit*.

Seznam dalších příkazů nástroje NuSMV je uveden v online dokumentu nazvaném *The NuSMV Interactive Shell Commands* [35].

### 6.3.2 Práce s grafickým rozhraním NuSMV

Pokud bychom chtěli využívat grafické uživatelské rozhraní gNuSMV, tak musíme provést různá nastavení a doinstalovat některé balíčky na námi používaný operační systém. Popis požadovaných částí a způsob jak vše nastavit nalezneme na těchto stránkách [33].

Pokud se nám podařilo vše nastavit a požadované balíčky nainstalovat, tak můžeme přejít k samotné práci s grafickým rozhraním gNuSMV. Než zadáme příkaz pro spuštění, tak se musíme v adresářové struktuře nacházet v adresáři, ve kterém je vše potřebné pro gNuSMV nainstalováno. Spuštění grafického rozhraní se provádí pomocí příkazu *python main.py*, který zadáme v příkazové řádce námi užívaného operačního systému. Pro zobrazení grafického rozhraní se používá interpret Python. Z tohoto důvodu se zadává příkaz *python*. Po zadání příkazu se nám zobrazí úvodní obrazovku gNuSMV. Tuto obrazovku můžeme vidět na obrázku č. 30.




Obrázek 30: Úvodní obrazovka gNuSMV.

Pokud již máme grafické rozhraní nástroje NuSMV spuštěné a vidíme obrazovku jako na orázku č. 30, tak můžeme přejít k samotné práci. Můžeme provádět verifikaci modelů, které jsme si předem vytvořili, nebo můžeme použít ukázkové příklady. Model systému otevřeme pomocí nabídky *Opens a new SMV file* a vybráním konkrétního souboru. Model se nám následně zobrazí v prostřední části v záložce *Modelling*. Pomocí tlačítka *Compile* provedeme kompilaci modelu a kontrolu případných chyb. Po provedení kompilace přejdeme na záložku *Properties*, kde můžeme provést pomocí tlačítka *Check!* verifikaci konkrétních formulí. Při stisku tlačítka, nám nástroj po chvilce zobrazí, zda verifikované formule jsou pravdivé či nikoliv. Grafické rozhraní dále nabízí uživateli několik možností, kterými můžeme blíže specifikovat chování nástroje NuSMV.

Nyní, když známe základní informace o SMV a NuSMV, tak můžeme přejít k hlavní části této práce. V následující části se budeme zabývat popisem nástroje SMV a tvorbou modelů systémů za použití tohoto nástroje.

# 7 Příkladová studie

Tato kapitola se více zaměřuje na popis nástroje SMV, strukturu modelu, kterou budeme využívat při tvorbě modelů různých systémů. Dále se kapitola zaměřuje na popis, jak probíhá konkrétní tvorba modelu systému v nástroji SMV od napsání kódu až po ověření specifikace. Kapitola taktéž popisuje, jak se nástroj SMV spouští a jaké další možnosti umožňuje. Po těchto informacích následuje část, která se již zabývá řešením konkrétních modelů různých systémů a jejich verifikováním.

## 7.1 Popis SMV

Popis SMV byl vypracován s využitím informací získaných z manuálu SMV [27]. Nejprve bude popsána modulární skladba programu popisujícího model systému. Syntaxe a sémantika základních příkazů nástroje SMV je uvedena v příloze A.

Model systému se skládá z jednoho nebo více modulů. Rozepsání do více modulů může zpřehlednit celkový model a umožňuje tak snadno popisovat systémy s opakujícími se prvky. Hlavním a povinným modulem SMV je *main*. V tomto modulu můžeme definovat instance dalších modulů. Základními datovými typy vstupního jazyka jsou binární číslo a skalár, ten je buď ve formě čísla ze zadaného konečného intervalu celých čísel, nebo ve formě výčtového typu. Definice proměnných a instancí modulů se uvádějí v sekci VAR.

### 7.1.1 Struktura modelu v SMV - obecně

- Definice modulů (nezáleží na pořadí)

  **MODULE** název modulu(výčet_vstupních_signálů)

- uvnitř modulu:

  sekce **VAR**

  - definice stavových proměnných
  - definice procesů (instanciace modulů)

  sekce **ASSIGN**

  - definice přechodové relace (nad stavovými proměnnými)

  sekce **DEFINE**

  - definice signálů

  sekce **SPEC**

  - definice vlastností v CTL (typicky v main)

### 7.1.2 Struktura modelu v SMV - prakticky

Nyní si ukážeme, jak vypadá struktura prakticky. Jedná se o ukázku programu v SMV reprezentujícího křižovatku, která je řešená v kapitole č. 7.10. Níže vidíme část vytvořeného modulu zajišťujícího změnu barevné signalizace.

```
MODULE svetlo(barva_jiny_sem)
VAR
      barva : {cer, ceroran, zel, oran};
ASSIGN
      init(barva) := cer;
      next(barva) := case
             ....
             esac;
```

Zde jsme viděli definici modulu, který je pojmenován *svetlo*. Tento modul obsahuje sekci VAR, kde jsou definovány stavové proměnné. V našem případě se jedná o světlo semaforu pojmenované *barva*. V sekci ASSIGN jsou definovány přechodové relace, tedy stavy, ve kterých se může světelná signalizace nacházet.

V další části máme zobrazenou ukázku hlavního modulu main. V sekci VAR je ukázáno vytvoření dvou instancí modulu *svetlo*.

```
MODULE main
VAR
      semafor1 : svetlo(semafor2.barva);
      semafor2 : svetlo(semafor1.barva);

SPEC AG!(sem_auto1.svetlo = zel & sem_chod1.svetlo_ch = zelena)
.....
```

Poslední částí modulu *main* je sekce SPEC, ve které je zapsána formule, která ověřuje, zda může nastat situace, kdy na semaforu pro chodce a vozidla bude zároveň svítit barva zelená.

## 7.2 SMV v krocích

Nyní, když víme jak jednotlivé základní části SMV jazyka používat, tak si můžeme říci, jak probíhá konkrétní tvorba modelu systému v SMV od napsání kódu až po ověření specifikace.

- **Zápis modelu**: napsání kódu v jazyce SMV do souboru *.smv
- **Čtení modelu**: čtení modelu ze vstupního souboru *.smv
- **Vytvoření hiearchie**: instanciování modulů a procesů

- **Tvorba modelu**: kompilování modelu do binárních rozhodovacích diagramů (BBD)
  - počáteční stavy, invarianty, přechodové relace a další.
- **Ověření**: ověřování specifikace vlastností

## 7.3 Spuštění SMV

Poslední částí, kterou nám zbývá popsat, než začneme vytvářet konkrétní modely systémů, je spouštění nástroje SMV. Jedná se o konzolovou (terminálovou) aplikaci, která se spouští v shellu. V případě jiné verze SMV, například NuSMV, můžeme využít i GUI rozhraní. V našem případě si ukážeme příkazy pro terminál. Obecně vypadá spouštěcí příkaz následovně.

smv [Volby] [Vstupní soubor]

Za *Volby* lze uvést velké množství příkazů. Zde si uvedeme jen některé. Ostatní je možné najít v literaturách, které se zabývají SMV.

Volby:

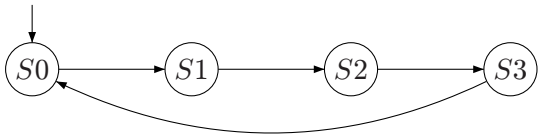
| | |
|---|---|
| -version | - verze SMV |
| -c | - nastavení velikosti cache pro BDD operace, defaultně je 32749 |
| -k | - nastavení velikosti klíčové tabulky pro BDD uzly |
| -m | - nastavení velikosti části cache pro BDD operace, které jsou používány jako méně nákladné (neiterativní) BDD operace |
| -int | - používání interaktivního módu |
| -r | - tisk výstupních statistik dosažitelných stavů |
| -AG | - ověřování pouze všeobecných CTL formulí |
| ... | - další formule |

## 7.4 Automat v SMV

Po definování struktury modelu v SMV a nastínění základní syntaxe a sémantiky, viz příloha A, vytvoříme náš první malý příklad. Následující podkapitoly nám osvětlí, jak vytvářet programy pro nástroj SMV, a jak následně verifikovat různé vlastnosti námi verifikovaných systémů.

V rámci prvního příkladu si zkusíme z uvedeného automatu vytvořit kód pro SMV a trošku si rozebereme jednotlivé části. Pro účely tohoto příkladu vezmeme automat, který je zobrazen na obrázku č. 31.

Na obrázku č. 31 vidíme automat, který je tvořen čtyřmi stavy $S0, S1, S2, S3$. Počátečním stavem je stav $S0$. Automat přijímá slova, jejichž zbytek po dělení čtyřmi je roven nule. Nyní si zkusíme tento automat převést do jazyka pro SMV.




Obrázek 31: Jednoduchý automat mod-4 čítač.

```
MODULE citac
VAR
      state: {S0, S1, S2, S3};
```

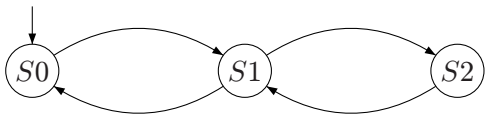
Nejprve si nadefinujeme hlavní modul, který nezveme *citac*. Dále následuje sekce *VAR*, ve které nadefinujeme symbolické stavy $S0, S1, S2, S3$. Po definici všech stavů přejdeme k určení počátečního stavu (init) a jednotlivých přechodů. To se provádí následujícím způsobem v sekci *ASSIGN*.

```
ASSIGN
      init(state) := S0;
      next (state) := case
            state = S0 : S1;
            state = S1 : S2;
            state = S2 : S3;
            1 : S0;
      esac;
```

Tímto způsobem jsme tedy vytvořili jednoduchý systém, se kterým můžeme dále pracovat v SMV. Uvedený příklad je v příloze na CD v adresáři *Příklady* pod názvem *automat.smv*.

## 7.5 Použití nedeterminismu v SMV

V následujícím příkladu si ukážeme použití nedeterminismu. Představme si následující nedeterministický automat, který je zobrazen na obrázku č. 32.




Obrázek 32: Jednoduchý nedeterministický automat.

Tento automat obsahuje jeden nedeterministický krok. Ten nastává v situaci, kdy chceme ze stavu $S1$ přejít do následujícího stavu. Je možno se rozhodnout mezi přechodem do stavu $S0$ ze stavu $S1$, nebo mezi přechodem do stavu $S2$ ze stavu $S1$. Prvotní

zápis bude podobný jako v předchozím příkladu. Jediný velký rozdíl bude v sekci *ASSIGN*, kde budou definovány oba přechody ze stavu $S1$. To se provádí pomocí *state = S1 : {S0, S2};*.

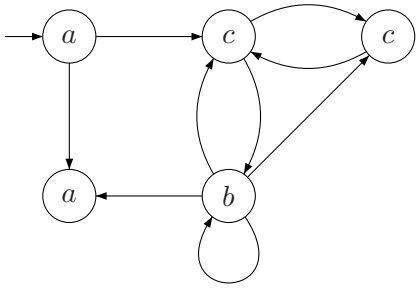
```
MODULE citac_2
VAR
      state: {S0, S1, S2};
ASSIGN
      init(state) := S0;
      next(state) := case
             state = S0 : S1;
             state = S1 : {S0, S2};
             state = S2 : S1;
             1 : S0;
             esac;
```

Tento příklad je uveden v příloze na CD v adresáři *Příklady* pod názvem *nedeter_aut.smv*.

## 7.6 Specifikace vlastností

Na tomto příkladu si ukážeme, jak převedeme počáteční, vymyšlený automat na konkrétní kód, nad nímž pak ověříme dvě základní vlastnosti, které je možno testovat pomocí nástroje SMV. Závěrem si ukážeme výpis a informace týkající se daného kódu, které nám poskytl nástroj SMV skrze příkazovou řádku.

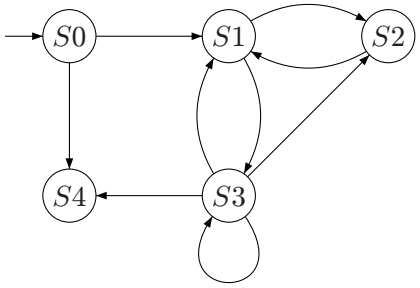
Pro začátek si uveďme jednoduchý automat, ze kterého budeme vycházet. Jedná se o vymyšlený automat, na kterém budeme demonstrovat způsob přepisu do kódu, který je stravitelný nástrojem SMV. Jednoduchý automat můžeme vidět na obrázku č. 33. Představme si, že jsme vnější pozorovatel, který pozoruje tento, dalo by se říci systém, a chceme si ověřit, zda existuje cesta, ve které bude stále platit vlastnost označená písmenem *a*, tedy *EG a*. Případně zjišťujeme, zda ve všech cestách stále platí námi ověřovaná vlastnost, *AG a*. Bohužel jako pozorovatel neznáme způsob jak si ověřit tyto vlastnosti systému. Zkusíme je tedy ověřit pomocí nástroje SMV.




Obrázek 33: První příklad - automat.

Náš automat je tvořen několika uzly, mezi nimiž máme různé neoznačené hrany. Hrany jsme si nepojmenovávali, neboť pro první příklad bude pro nás jejich pojmenování

nepodstatné. Jednotlivé uzly jsou pojmenovány následovně pomocí symbolů: *a*, *b*, *c*. První uzel pojmenovaný *a*, který je vlevo nahoře, má navíc označení počátečního uzlu. Počáteční uzel tedy označujeme šipkou, která směřuje do uzlu.
Tento automat a jeho uzly si trošku upravíme do podoby, ze které budeme vycházet a následně tvořit kód pro SMV. Jednotlivé uzly si pojmenujeme jako S0, S1, S2, S3 a S4. Toto označení nám poslouží v dalších krocích a zjednoduší nám tvorbu kódu. Původní označení počátečními písmeny abecedy platí i nadále. Nové označení symboly S0 až S4 lze vidět na obrázku 34.




Obrázek 34: Pojmenování symboly S0, S1, S2, S3, S4.

Nyní se podíváme na to, jak tento automat přepsat do jazyka SMV. Nejprve musíme definovat jednotlivé stavy a to jsou stavy S0, S1, S2, S3, S4. Definování jednotlivých stavů se provádí v sekci VAR v hlavním modulu MODULE main, následujícím zápisem:

```
MODULE main
VAR
      state: {S0, S1, S2, S3, S4};
```

Pokud jsme nadefinovali jednotlivé uzly, tak přejdeme k definování počátečního uzlu a k definování přechodů mezi jednotlivými stavy. Toto vše je možno provést v sekci ASSIGN. Pro definování hran mezi uzly využijeme *case operátoru*. Následující výpis, nám zobrazuje, jakým způsobem jsou jednotlivé části zapsány.

```
ASSIGN
      init (state) := S0;
      next (state) := case
             state = S0 : {S1, S4};
             state = S1 : {S2, S3};
             state = S2 : {S1};
             state = S3 : {S1, S2, S3, S4};
             state = S4 : S4;
             esac;
```

Máme tedy vytvořeny jednotlivé stavy a hrany mezi těmito stavy s tím, že jsme si určili i počáteční stav. Nyní nám zbývá dodefinovat, které stavy budou jak pojmenovány. Tedy naše výše zmíněné pojmenování *a*, *b*, *c*. To se provádí v sekci DEFINE pomocí následujícího zápisu.

```
DEFINE
      a := (state = S0 | state = S4);
      b := (state = S3);
      c := (state = S1 | state = S2);
```

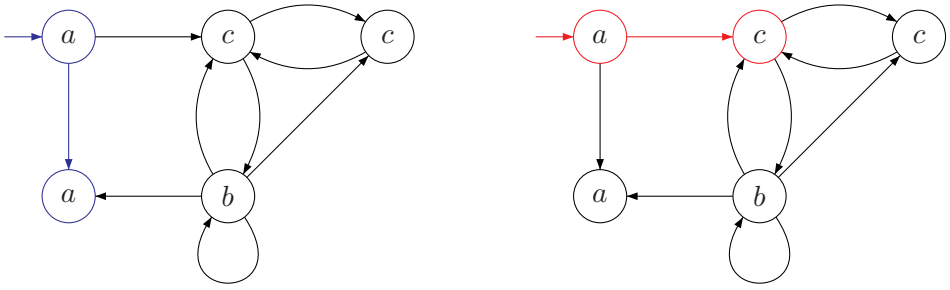
V této chvíli máme vytvořen automat jako na obrázku č. 33. Nyní, když je automat vytvořen a vše je v pořádku, je možno přejít na část, ve které budeme ověřovat jednotlivé vlastnosti tohoto automatu. Zápis jednotlivých vlastností se provádí pomocí specifikace SPEC a zápisu vlastnosti. Tento způsob můžeme vidět níže. Tuto část je nutno dopsat do dokumentu, ve kterém jsme si zapisovali jednotlivé části kódu našeho automatu. Definujeme tedy následující vlastnosti:

```
SPEC EG a
SPEC AG a
```

Vlastnost *SPEC EG a*, která je specifikovaná v našem příkladě znamená, že existuje cesta, ve které platí *a*. Naproti tomu vlastnost *SPEC AG a* znamená, že ve všech cestách je splněno *a*.

Nyní stačí provést verifikaci pomocí nástroje SMV. Jestliže ji provedeme, tak zjistíme, že nám nástroj SMV dal dva výsledky. Tedy odpovědi na naše otázky týkající se vlastností *EG a* a *AG a*. První vlastnost *EG a* je true, neboli pravdivá. Existuje totiž cesta, ve které je *a* dosažitelné. Naproti tomu neexistuje cesta taková, že by ve všech cestách platilo *a*. Proto je daná vlastnost false, neboli nepravdivá. Vyhodnocení obou vlastností nástrojem SMV nám ukazuje následující výpis.

```
-- specification EG a is true
-- specification AG a is false
```



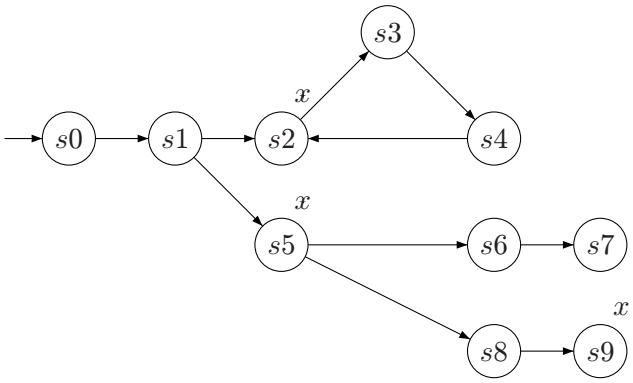

Obrázek 35: Vlastnost EG (true), protipříklad vlastnosti AG (false)

Graficky znázorněné řešení vlastnosti *EG a* můžeme vidět na obrázku č. 35 vlevo. Na tomto obrázku vidíme vpravo také protipříklad, který nám vlastnost *AG a* nesplňuje. Z tohoto důvodu je odpověď na druhou vlastnost *false*. Příklad je možné si prohlédnout také v příloze na CD v adresáři *Příklady* pod názvem *specifikace.smv*.

## 7.7 Specifikace vlastností, druhý příklad

V předešlém příkladu jsme si detailně popsali jak vytvořit konkrétní automat a následně verifikovat jeho vlastnosti. Možné formule specifikace a jejich význam je uveden v podkapitole temporální logiky. Jedná se o podkapitolu s označením 3.2. Tato podkapitola se zabývá CTL formulemi a jejich popisem. V následujícím příkladu si zvolíme automat a na něj, pomocí specifikace vlastností, aplikujeme některé vybrané formule.

Nyní, aniž bychom si opět popisovali celý postup, provedeme konstrukci a verifikaci zvoleného automatu. Automat, který chceme verifikovat je na obrázku č. 36.




Obrázek 36: Automat.

Pro tento zvolený automat si nyní vytvoříme model pomocí jazyka SMV. Model můžeme vidět dále.

```
MODULE main
VAR
      state : {s0, s1, s2, s3, s4, s5, s6, s7, s8, s9};
ASSIGN
      init (state) := s0;
      next(state) := case
             state = s0 : s1;
             state = s1 : {s2, s5};
             state = s2 : s3;
             state = s3 : s4;
             state = s4 : s2;
             state = s5 : {s6, s8};
             state = s6 : s7;
             state = s7 : s7;
             state = s8 : s9;
             state = s9 : s9;
             esac;
DEFINE
      x := (state = s2 | state = s5 | state = s9);
```

Víme tedy, jak náš automat vypadá, víme jak jej převést do jazyka SMV a nyní zbývá jen ověřit konkrétní vlastnosti. Budeme chtít znát odpověď na následující formule.

```
SPEC x
SPEC AF x
SPEC EF x
SPEC EG x
SPEC AG x
SPEC E[(!x) U (state = s2)]
SPEC A[(!x) U (state = s2)]
```

Čtenář, který zná význam jednotlivých symbolů a daného zápisu, si sám odvodí, která vlastnost je či není pravdivá. Pro kontrolu uvedu správné odpovědi. Vlastnost *SPEC x* je nepravdivá. Tato atomická formule neplatí již v prvním stavu. Vlastnosti *SPEC AF x* a *SPEC EF x* platí. Neboť ve všech cestách eventuálně platí *x* a také víme, že existuje nějaká cesta, ve které platí *x*. Vlastnost *SPEC EG x* pravdivá není, neboť v našem automatu není cesta, ve které by stále platilo *x*. V případě, že není ani jedna cesta, ve které by stále platilo *x*, pak nemůže nastat případ, že by v automatu byly všechny cesty takové, že v nich platí stále *x*. Z tohoto důvodu neplatí ani vlastnost *SPEC AG x*. Na závěr nám zbývají poslední dvě vlastnosti, které nám říkají následující. Existuje cesta taková, ve které platí určitá část, dokud platí i část druhá. V druhém případě, kdy je zápis $A[A \cup B]$ se jedná o všechny cesty. První vlastnost *SPEC E[(!x) ∪ (state = s2)]* tedy pravdivá je a druhá vlastnost *SPEC A[(!x) ∪ (state = s2)]* pravdivá není. Uvedený příklad je v příloze na CD v adresáři *Příklady* pod názvem *specifikace_2.smv*.

### 7.7.1 Symbolický přístup k verifikaci CTL

Již dříve jsme si řekli, že specifikace vlastností modelu systému je tvořená klíčovým slovem SPEC a CTL formulí, která vyjadřuje konkrétní verifikovanou vlastnost. Nyní nám vyvstává otázka, jak dochází k verifikování námi zapsaných CTL formulí. Jistě uznáme za pravdivé, že pokud známe pro každý stav modelu systému platnost formulí $\varphi$ a $\psi$, pak jistě snadno odvodíme i platnost formulí $\neg\varphi$, $\varphi \vee \psi$, EX$\varphi$ a dalších. Idea algoritmu pro CTL model checking je následující.

- Máme dánu Kripkeho strukturu $M$ (viz kapitola č. 4.1) $< S, S_0, R, L >$ a formuli $\varphi$.
- Spočítáme značkovací funkci pojmenovanou $label : S \rightarrow 2^{\varphi}$. Tato značkovací funkce o každém stavu $s \in S$ Kripkeho struktury $M$ řekne jaké podformule námi uvedené formule $\varphi$ platí v konkrétním stavu.
- Platí následující: $S_0 \models \varphi \Leftrightarrow \varphi \in label(s)$.
- Funkci *label* vypočteme postupně pro jednotlivé podformule $\varphi$. Výpočet provádíme tak, že začínáme od nejjednodušších podformulí a pokračujeme k těm složitějším. Závěrem bychom měli dojít až k podformuli $\varphi$.

Z této idey algoritmu nám vyplývá několik myšlenek. Například si budeme kompaktně pamatovat (budovat) množiny stavů, ve kterých platí jednotlivé podformule verifikované CTL formule. Explicitní počítání funkce *label* nahradíme manipulací s těmito kompaktními reprezentacemi. V předešlých kapitolách jsme se zmínili o binárních rozhodovacích diagramech a práci s těmito diagramy. Pokud bychom tedy chtěli realizovat algoritmus řešící symbolický přístup k verifikaci CTL, pak jednotlivé množiny stavů jsou udržovány pomocí OBDD struktur. Výchozím bodem jsou OBDD pro jednotlivé AP neboli množiny atomických výroků. Podle struktury formule je vypočítáváno OBDD pro jednotlivé podformule. Závěrem je otestována přítomnost iniciálního stavu v množině stavů, splňující verifikovanou formuli.

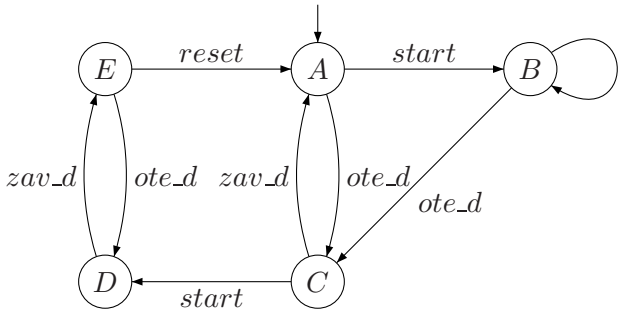
## 7.8 Mikrovlnná trouba

Jednou z dalších oblastí, ve které je možné využít možnosti nástroje SMV, je oblast zabývající se verifikováním modelů systémů, které představují různé domácí, či kuchyňské spotřebiče (sporák, mikrovlnná trouba a další). Nástroj SMV, se taktéž používá pro verifikování různých systémů například pro ovládání garážových vrat, bezpečnostních systémů, systémů zabývajících se udržováním klimatických podmínek ve sklenících, či systémů pro ovládání výtahů. Při práci s nástrojem SMV jsem se zaměřil na první uvedenou oblast. Oblast představující kuchyňské spotřebiče. Inspiroval jsem se modelem, který je online uveden zde [28] a následně jsem vytvořil a provedl verifikaci modelu systému představujícího mikrovlnou troubu.

V této části si ukážeme jak pomocí nástroje SMV ověřit několik vlastností týkajících se modelu mikrovlnné trouby. Chování modelu mikrovlnné trouby je následující. Mikrovlnná trouba je vybavena dvířky. Tato dvířka je možné otevírat a zavírat. V případě vložení potravin určených pro tepelné zpracování do mikrovlnné trouby a při zavřených dvířkách, je možné pomocí stisku tlačítka *START* spustit režim pečení. Tento režim je ukončen otevřením dvířek a vyjmutím potravin. V případě, že byla dvířka otevřena a je požadováno opětovné pečení, musí se opět stisknout tlačítko *START*. Jestliže uživatel má otevřená dvířka a stiskl například omylem tlačítko *START*, tak se mikrovlnná trouba dostane do stavu, který představuje chybu. Není možné provádět pečení při otevřených dvířkách. Z tohoto stavu se dostaneme tak, že dvířka zavřeme a následně stiskneme tlačítko *RESET*.

Model mikrovlnné trouby lze popsat pomocí automatu. Automat, který představuje chování našeho modelu je zobrazen na obrázku č. 37.

Stavy označené velkými písmeny $A, \ldots, E$ představují možné stavy mikrovlnné trouby. Stav označený jako $A$ je stav počáteční, který představuje stav, kdy jsou: zavřená dvířka, není aktivní pečení a nenastala chyba. Při stisknutí tlačítka *(start)* se dostaneme do stavu, který je označen jako $B$. Tento stav představuje: mikrovlnná trouba je zapnutá, peče se a taktéž nenastala chyba. Po otevření dvířek *(ote_d)* přejdeme do stavu $C$, kdy jsou: otevřená dvířka, nepeče se a taktéž nenastala chyba. V případě, že jsme z počátečního stavu, kdy nebyla otevřená dvířka, přešli do stavu $C$ otevřením dvířek *(ote_d)*, tak můžeme pokračovat buď zpět do počátečního stavu (dvířka zavřeme, *zav_d*), nebo při stisku tlačítka *start* se dostaneme do stavu $D$, který představuje: dvířka jsou otevřená, nepeče se a nastala




Obrázek 37: Automat představující chování mikrovlnné trouby.

chyba. Z tohoto chybového stavu se dostaneme zavřením dvířek *zav_d* a následným stiskem tlačítka *reset*.

V předešlých příkladech jsme přesně popisovali v nástroji SMV jednotlivé přechody námi uvedených automatů. Nyní si ukážeme trošku odlišný způsob. Budeme zapisovat počáteční stavy určitých veličin *(peče se, chyba)* a přechod do dalších stavů vlivem splnění určitých podmínek. Kód v jazyce SMV můžeme vidět na následujícím výpisu.

```
MODULE main

VAR
      dvirka : {otev, zav};
      pece_se : boolean;
      chyba : boolean;
      start : boolean;
      reset : boolean;

ASSIGN
      init(chyba) := 0;
      next(chyba) := case
            start & (dvirka = otev) : 1;
            (dvirka = zav) & reset : 0;
            1 : chyba;
            esac;

      init(pece_se) := 0;
      next(pece_se) := case
            start & dvirka = zav : 1;
            dvirka = otev : 0;
            1 : pece_se;
            esac;
```

```
SPEC EF(pece_se)
SPEC EF(chyba)
SPEC AG((start & (dvirka = otev)) -> AX(chyba))
SPEC AG((dvirka = otev) -> AX (!pece_se))
SPEC AG((start & (dvirka = zav))  -> EF(pece_se))
```

V sekci VAR definujeme typ jednotlivých proměnných. Většina proměnných má boolean hodnotu, vyjma proměnné *dvirka*, která je určená výčtem. V sekci ASSIGN provedeme nastavení počátečních hodnot *(init)* proměnných *chyba* a *pece_se*. Po počátečním nastavení se zapíší možnosti následujícího stavu konkrétních proměnných. Chybový stav může nastat v případě, že je stisknuto tlačítko *start* a jsou otevřená dvířka *dvirka = otev*. Tento stav představující chybu systému opouštíme, jestliže uživatel dvířka zavře a následně stiskne tlačítko *reset*. Poté nastavíme ještě následující stav, který nám bude říkat, kdy se provádí pečení. Péct je možno v situaci, kdy jsou dvířka uzavřená a bylo stisknuto tlačítko *start*. Pokud bychom v průběhu této činnosti dvířka otevřeli, tak musí dojít k přerušení pečení. V závěru již jen dopíšeme několik specifikací vlastností našeho modelu mikrovlnné trouby. Můžeme například ověřovat, zda může nastat někdy činnost představující pečení, nebo zda k této činnosti může dojít při otevřených dvířkách, případně zda dojde v nějaké situaci k chybě. Vlastností, které můžeme nad tímto modelem ověřovat je možno uvést více. V našem příkladu jsou uvedeny jen některé vybrané vlastnosti. Příklad modelu mikrovlnné trouby je uveden v příloze na CD v adresáři *Příklady* pod názvem *mi_trouba.smv*.

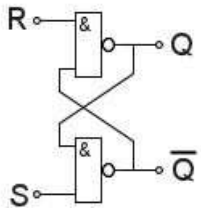
V případě řešení modelů systémů podobných výše uvedenému lze k ověřování chování použít nástroj SMV. Nástroj SMV je však předurčen spíše k ověřování systémů založených na logických prvcích a hardwarových částech. Jedná se například o digitální číslicové obvody.

## 7.9 Hardwarové obvody v SMV

V této části bude ukázáno jak převést jednoduchý logický obvod na model, který bude následně verifikován v nástroji SMV. Následující část jsem vypracoval s využitím informací uvedených v online dokumentu *Hardware verification using model checking* [29].

V předešlých příkladech jsme si ukázali jak popsat různé automaty, případně model kuchyňského spotřebiče. V této části si ukážeme jak je možno popsat systém, který je tvořen hardwarovými součástkami, například NAND, NOR, případně dalšími. Jednoduchý obvod můžeme vidět na obrázku č. 38.

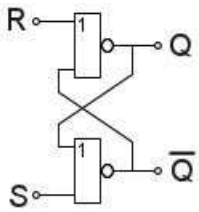
Na obrázku č. 38 vidíme RS klopný obvod. Klopné obvody jako takové jsou nejjednodušší digitální paměťové obvody. Z hlediska vztahu mezi vstupními a výstupními proměnnými rozdělujeme klopné obvody do čtyř typů. Jedná se o klopné obvody RS, JK, T a D. Klopné obvody jsou dále děleny na synchronní a asynchronní. Asynchronní klopné obvody mohou změnit hodnotu výstupní proměnné nezávisle na hodinovém signálu. Asynchronní klopné obvody nemají hodinový vstup. Synchronní klopné obvody jsou takové obvody, které mohou změnit hodnotu výstupní proměnné jen při úrovni H nebo L hodinových impulzů.




Obrázek 38: R-S klopný obvod (hradla NAND)

### 7.9.1 Klopný obvod RS

RS klopný obvod je z uvedených typů ten nejjednodušší klopný obvod. Klopný obvod RS má ve svém základním provedení dva vstupy a to vstup S (SET) a R (RESET). Česky bychom řekli nastavení a nulování. Vstupem SET nastavíme na výstup klopného obvodu, logickou jedničku a vstupem RESET logickou nulu. Obvod si námi zadaný stav pamatuje, a to tak dlouho dokud není námi změněn. Klopný obvod RS může být vytvořen několika způsoby a z různých hradel (například NAND, NOR) a podle toho se také tento klopný obvod chová. Klopný obvod RS z hradel NAND jsme viděli již na obrázku č. 38. RS klopný obvod tvořený hradly NOR můžeme vidět na obrázku č. 39.




Obrázek 39: R-S klopný obvod (hradla NOR)

Pravdivostní tabulku RS klopného obvodu, který je tvořen hradly NAND můžeme vidět dále. Druhá tabulka, která je zde zobrazená, je pravdivostní tabulka RS klopného obvodu, který je tvořen hradly NOR.

### 7.9.2 Tabulka obvodu RS z hradel NAND

| R | S | Q | $\bar{Q}$ | Poznámka |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | Zakázaný stav |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | |
| 1 | 1 | $Q_n$ | $Q_n$ | Zůstává předchozí stav |

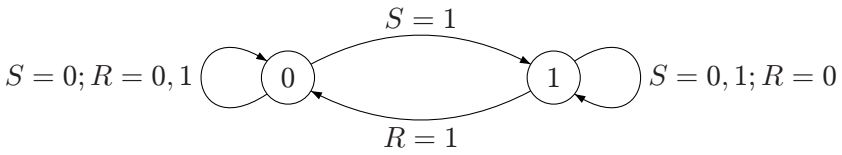
### 7.9.3 Tabulka obvodu RS z hradel NOR

| R | S | Q | $\bar{Q}$ | Poznámka |
|---|---|---|---|---|
| 0 | 0 | $Q_n$ | $Q_n$ | Zůstává předchozí stav |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | |
| 1 | 1 | 0 | 0 | Zakázaný stav |

Nyní si popíšeme RS klopné obvody. Klopné obvody RS mají zavedenu stoprocentní stejnosměrnou zpětnou vazbu. Tato zpětná vazba je z výstupu jednoho hradla na vstup hradla druhého. Jejich vstupy jsou označeny jako R a S a výstupy jako Q a $\bar{Q}$, který je k výstupu Q inverzní. To platí kromě zakázaného stavu, ve kterém jsou oba výstupy ve stejném stavu, log. 1 pro NAND a log. 0 pro NOR.

Klopný obvod tvořený hradly NAND je ve výchozím stavu, když je na obou vstupech log. 1. Pokud na SET přivedeme log. 0, hradlo se překlopí tak, že na výstupu Q bude log. 0. Pokud tam již log. 0 byla, potom obvod v tomto stavu zůstane. Pokud se vrátíme do výchozího stavu, tak informace, která byla na výstupech, na nich bude i nadále. Tato informace zde bude tak dlouho, dokud nepřivedeme log. 0 na RESET, potom se hradlo překlopí a výstupní informace se změní. Obdobně tomu je i u klopného obvodu s hradly NOR. Pouze je zaměněn výchozí stav. V něm jsou zde oba vývody v log. 0. Zakázaný stav je, když je na obou vstupech log. 1.

Chování klopného obvodu je možné také popsat pomocí automatu. Automat, který představuje chování klopného obvodu RS je zobrazen na obrázku č. 40. Stavy automatu představují stavy RS klopného obvodu. Hodnota dvou stavů automatu (0, 1) je hodnota RS klopného obvodu na výstupu Q.




Obrázek 40: Automat představující chování RS klopného obvodu.

Nyní již víme jaké je schéma klopného obvodu RS a víme, jak by se měl tento obvod chovat. Nyní nám zbývá popsat model RS klopného obvodu v SMV jazyce.

Víme, že klopný obvod RS je tvořen dvěma hradly NAND (resp. NOR). V následujícím případě budeme uvažovat použití hradel NAND. Vytvoříme si tedy modul pojmenovaný *Nand*, který bude zajišťovat stejnou funkcionalitu jako hradlo NAND. Tento modul můžeme vidět dále.

```
MODULE Nand(vstup_1, vstup_2)

VAR
      vystup : boolean;
```

```
ASSIGN
      init(vystup) := 1;
      next(vystup) := !(vstup_1 & vstup_2);
```

Modul *Nand* má dva vstupní parametry. Jsou jimi *vstup_1, vstup_2*, které představují dva vstupy hradla. V sekci VAR je definována proměnná *vystup*, která představuje výstup hradla. Dále v sekci ASSIGN definujeme počáteční a následující hodnotu proměnné *vystup*. Nastavení následující hodnoty proměnné *vystup* určíme tak, že provedeme negaci logického součinu dvou vstupních proměnných *!(vstup_1 & vstup_2)*. Nyní tedy máme vytvořený modul, který se chová přesně jako logický člen NAND. V druhém modulu, který si pojmenujeme *RSobvod* budeme řešit již kompletní RS klopný obvod. Již dříve jsme si řekli, že RS klopný obvod je tvořen dvěma hradly Nand, které jsou mezi sebou propojené. Modul, který bude řešit RS klopný obvod je zobrazen zde:

```
MODULE RSobvod(vstup_r, vstup_s)
VAR
      prvni_nand : Nand(vstup_r, druhy_nand.vystup);
      druhy_nand : Nand(vstup_s, prvni_nand.vystup);
```

Tento modul má dva vstupní parametry. Těmito parametry jsou *vstup_r, vstup_s*, které představují vstup R (Reset) a S (Set). V sekci VAR máme vytvořeny dva hradla Nand, ze kterých se RS klopný obvod skládá. Výstup z jednoho hradla, který je vstupem druhého hradla je vytvořen tak, že budeme využívat proměnné *vystup* konkrétního hradla Nand. Pomocí těchto dvou modulů máme vytvořen RS klopný obvod. Tyto obvody bývají součástí různých hardwarových obvodů. Díky možnosti převést jednotlivé složité hardwarové části na menší bloky, které jsou tvořené základními prvky (například hradla AND, NAND, OR, atd), je možné verifikovat různé modely skutečných hardwarových systémů. Příklad zobrazující model klopného obvodu je v příloze na CD v adresáři *Příklady* pod názvem *rs_obvod.smv*.

## 7.10 Křižovatka

V následujícím příkladu se zaměříme na řešení modelu, který představuje reálnější systém z našeho života. Zkusíme si namodelovat a poté ověřit vlastnosti systému, který bude představovat křižovatku. Křižovatku tvořenou různými semafory, které budou obsluhovat různé směry, řídit dopravu a zabraňovat kolizi. V následujících částech budou uvedeny postupy tvorby modelu reprezentujícího systém křižovatky.

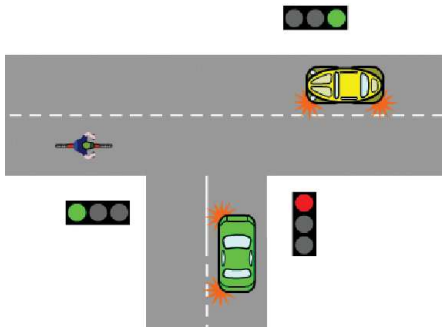
Pokusme se tedy namodelovat model systému, který je zobrazen na obrázku č. 41. Jedná se o model křižovatky, která má tři směry. Na obrázku je horizontálně zobrazena hlavní jedno proudá silnice, na které jsou umístěny dva semafory. Tyto semafory pracují ve stejném režimu. Tedy na obou semaforech bude vždy svítit stejná barva. Vertikálně je na obrázku zobrazena vedlejší silnice. I na této silnici, ačkoliv se jedná pouze o silnici vedlejší, je umístěn semafor.

Chování tohoto modelu reálného systému je následující. Na všech semaforech bude po spuštění svítit barva červená. Poté dojde ke změně signalizace. V případě, že na semaforech, které obsluhují hlavní silnici, bude svítit zelená, pak na semaforu, který obsluhuje vedlejší silnici, bude svítit barva červená. V opačném případě bude na semaforech umístěných u hlavní silnice svítit barva červená a na semaforu, který je umístěn u silnice vedlejší, bude svítit barva zelená. Nesmí nastat situace, kdy by na všech semaforech svítily zároveň stejné barvy, vyjímaje počátečního stavu. Změny signalizace budou probíhat, tak jak je popsáno v následujícím odstavci. Bude-li umožněn průjezd křižovatkou na hlavní silnici, pak se při odbočování dává navíc přednost protijedoucím dopravním prostředkům. V obrázku č. 41 je naznačená situace, kdy vozidlo na hlavní silnici odbočuje vlevo. V případě, že má zajištěn semaforem volný průjezd, tak musí dát přednost protijedoucímu cyklistovi, který jede rovně. Nejprve tedy projede křižovatkou cyklista, poté odbočující vozidlo. Po změně signalizace může křižovatku opustit i vozidlo odbočující z vedlejší silnice.

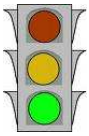
Semafory, které jsou umístěny na této křižovatce, mají tři základní barvy - červená, oranžová a zelená. Systém změny signalizace je evropský. Pokud je potřeba, aby konkrétní směr byl zastaven, tak na semaforu svítí barva červená. Účastníkům dopravy je signalizováno, že musí stát na místě před semaforem, případně na vyznačeném místě (bílé čáry na vozovce). Jestliže je směr zastaven a nastala doba, kdy se mají účastníci dopravy připravit na změnu, tedy bude následně směr průjezdný, pak je tato situace signalizována současným rozsvícením barev červená a oranžová. Účastníci dopravy se připravují na opuštění vyhrazeného místa. Dalším stavem, který následuje je stav, kdy je účastníkům dopravy povolen bezpečný vjezd do křižovatky. Tento stav je signalizován pomocí rozsvícení zelené barvy. Následuje přechod ze stavu umožňujícího bezpečný průjezd křižovatkou do následného stavu, který bude informovat o tom, že dojde k zastavení konkrétního směru. Toto je signalizováno rozsvícením barvy oranžové. Nyní se účastníci dopravy musí připravit na to, že jejich směr bude zastaven. Buď zastaví své vozidlo na vyznačeném místě, nebo v případě vysoké rychlosti, a nemožnosti bezpečně zastavit na vyznačeném místě, mohou ještě křižovatkou projet. Je však kladen důraz na to, aby nebylo zneužíváno tohoto stavu k volnému projíždění křižovatky. Typ semaforu, který je umístěn na hlavní i vedlejší silnici je zobrazen na obrázku č. 42.

V našem modelu se budeme snažit o zajištění změny světel na semaforech tak, aby nedošlo ke kolizi účastníků provozu. Semafor a změnu signalizace na něm, lze simulovat automatem, který je tvořen čtyřmi stavy. Na obrázku č. 43 jsou zobrazeny dva automaty. První automat *Sem1* bude představovat chování semaforu na hlavní silnici. Druhý automat *Sem2* bude představovat chování semaforu na silnici vedlejší. Jsou zobrazeny jen dva automaty, neboť semafory na hlavní silnici mají stejné chování, signalizují vždy stejné barvy.

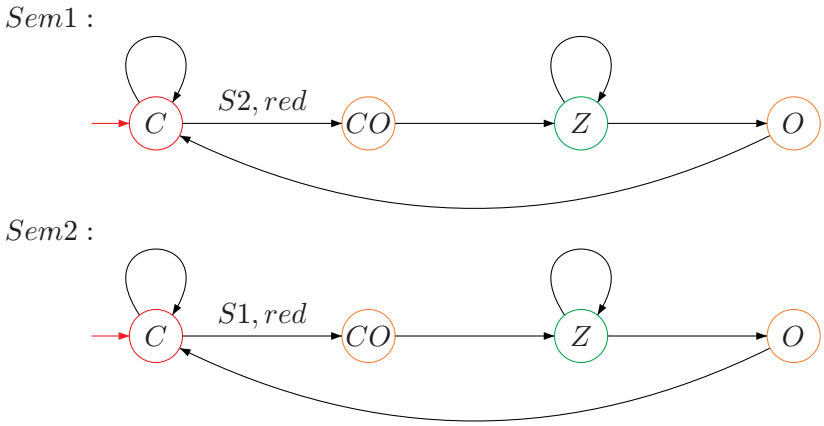
Automat představující chování systému má čtyři stavy. Tyto stavy představují barevné signalizace na semaforu. Stav označený písmenem *C* představuje červenou barvu, stav označený jako *CO* představuje současně rozsvícení barev: červená a oranžová. V případě stavu *Z* se jedná o zelenou barvu. Posledním stavem je stav *O*, který představuje oranžovou barvu. V popisu modelu je uvedeno, že počátečním stavem semaforů je stav, kdy



Obrázek 41: Křižovatka



Obrázek 42: Semafor




Obrázek 43: Automaty představující semafory.

svítí červená. Proto je stav *C* označen jako počáteční. V případě, že na druhém semaforu svítí taktéž červená barva, pak můžeme přejít do následujícího stavu pojmenovaného *CO*. Jedná se o situaci znázorněnou pomocí *SX, red*, kde *X* reprezentuje číslo dalšího semaforu. Ze stavu *CO* je veden přechod do stavu *Z*. Máme v úmyslu řešit systém, kdy bude svítit na jednom semaforu červená a na druhém semaforu dojde ke změnám až na zelenou a následně zpět na červenou. Nedojde tedy ke kolizi vozidel ani v případě, kdy by jedno vozidlo na oranžovou již vjíždělo do křižovatky a druhé na signalizaci červené s oranžovou nestačilo zastavit. Z důvodu, že se semafor může nacházet určitou chvíli ve stavu, kdy svítí zelená, tak je uvedená smyčka i nad stavem *Z*. Vozidla mohou projíždět,

aniž by ihned došlo ke změně stavu. Ze stavu *Z* následuje přechod do stavu *O*. Dochází tedy ke změně signalizace ze zelené na oranžovou. Účastníci provozu se připravují na to, že bude v brzké době provoz v daném směru zastaven. Ze stavu *O* se vracíme do počátečního stavu a celý průběh se opakuje. Jedná se totiž o systém, který běží tzv. do nekonečna.

Nyní, když máme představu, jak se chová konkrétní semafor, a máme toto chování znázorněno pomocí automatu, tak můžeme přejít k přepisu automatu do jazyka SMV. Pro vyřešení tohoto systému, nám postačí dva moduly. Z dvou modulů bude jeden modul hlavní (*main*), který bude zajišťovat volání obou semaforů. Druhý modul bude řešit přechody jednotlivých stavů automatu neboli změnu světel na semaforu. Napišme si tedy nejprve modul zajišťující změnu světel. Stavy cer (červená), ceroran (červená-oranžová), zel (zelená), oran (oranžová) představují jednotlivé kombinace světel na semaforu. To, jak bude tento modul vypadat, můžeme vidět dále.

```
MODULE svetlo(barva_jiny_sem)
VAR
      barva : {cer, ceroran, zel, oran};
ASSIGN
      init(barva) := cer;
      next(barva) := case
            barva = cer & barva_jiny_sem = cer :
                                  {ceroran, cer};
            barva = ceroran : zel;
            barva = zel : {zel, oran};
            barva = oran : cer;
            1 : barva;
            esac;
```

Tímto bychom měli vytvořený modul, pojmenovaný *svetlo*, který nám zajistí jednotlivé přechody světelných kombinací. Tomuto modulu je předáván parametr *barva_jiny_sem*, který nám říká, jakou signalizaci má právě jiný semafor. Tedy semafor ze směru, kde má být opačná signalizace. Modul obsahuje definici proměnných. Zde je definována proměnná *barva*, která nabývá hodnot *cer, ceroran, zel, oran*. Tyto hodnoty představují konkrétní barvy na semaforu. V uvedeném modulu je dále nastavená počáteční barva a to barva červená *cer*. Máme zajištěno, že při aktualizaci bude svítit na semaforu červená barva, jak tomu bylo požadováno v popisu modelu. Následně je nastavena změna barev. Pokud je nyní barva červená *cer* a obsahuje-li předaný parametr hodnotu také červenou *cer*, pak dojde ke změně a na semaforu bude svítit červená a oranžová zároveň *ceroran*. Další příkaz, který nám definuje následující stav, nám říká, že v případě, že svítí červená a oranžová zároveň *ceroran*, tak dojde ke změně na zelenou *zel*. Takto dochází ke změně i v dalších stavech.

Můžeme si všimnout, že jednotlivé přechody mezi konkrétními stavy probíhají jen za určitých podmínek, které musí být splněny, nebo pomocí nedeterminismu. Nikde zde není uvedena žádná proměnná, která by představovala čas. Toto omezení vyplývá přímo z jazyka SMV. V jazyce SMV totiž nelze definovat žádným způsobem čas.

Máme tedy modul, který nám zajistí změnu barev. Nyní vytvoříme hlavní modul. V hlavním modulu *main* budeme vytvářet konkrétní semafory. Hlavní modul můžeme vidět níže.

```
MODULE main
VAR
      semafor1 : svetlo(semafor2.barva);
      semafor2 : svetlo(semafor1.barva);
```

Hlavní modul je vždy pojmenován jako *main*. V sekci, kde bychom měli deklarovat proměnné, definujeme dvě instance. Těmito instancemi jsou *semafor1* a *semafor2*. Toto vytvoření instancí si můžeme snadno splést s definováním proměnných. Musíme dávat pozor na správný zápis. Instanci vytvoříme pomocí jejího jména, za kterým následuje dvojtečka a poté název modulu, který voláme. Tedy *jméno instance : název modulu (parametry);*, konkrétně pak *semafor2 : svetlo(semafor1.barva);*. Vytvořili jsme tedy dvě instance semaforů.

V rámci hlavního modulu se většinou píší i specifikace vlastností, které chceme nad daným modelem ověřovat. Specifikace vlastností se však mohou psát i do jiných modulů. Do hlavního modulu se píší hlavně specifikace vlastností celého systému. Zápis specifikace vlastností můžeme vidět zde.

```
-- Safety properties
SPEC AG!(semafor1.barva = zel & semafor2.barva = zel)
SPEC AG(semafor1.barva = zel -> !(semafor2.barva = ceroran|
semafor2.barva = oran|semafor2.barva = zel))
SPEC AG(semafor2.barva = zel -> !(semafor1.barva = ceroran|
semafor1.barva = oran|semafor1.barva = zel))

-- Livenes properties
SPEC EG(semafor1.barva = zel & semafor2.barva = cer)
SPEC EG(semafor1.barva = cer & semafor2.barva = cer)
```

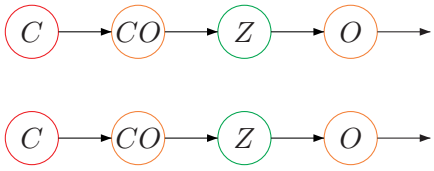
Při ověřování vlastností tohoto sytému nám SMV poskytne odpovědi na konkrétní specifikace. První vlastnost je označena za nepravdivou a zobrazený protipříklad nám ukazuje, že tomu tak opravdu je.

```
state 1.1:
semafor1.state = cer
semafor2.state = cer

state 1.2:
semafor1.state = ceroran
semafor2.state = ceroran
```

```
state 1.3:
semafor1.state = zel
semafor2.state = zel
```

Druhá a třetí vlastnost je taky označena za nepravdivou a to ze stejného důvodu, jako tomu bylo v případě první vlastnosti. Opět docházelo ke stejným kombinacím stavů systému. Na semaforech vždy svítily stejné barvy. Barevnou signalizaci na uvedených semaforech můžeme vidět na obrázku č. 44.



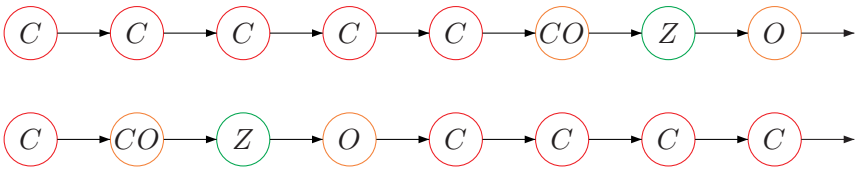

Obrázek 44: Barevná signalizace na semaforech.

Jak tedy zajistit, aby nám oba semafory nezobrazovaly tytéž světelné sekvence? Postačí k tomu doplnit do hlavního modulu klíčové slovo *process* v definici instance. Použitím tohoto klíčového slova *process* v hlavním modulu zajistíme, že se provede nedeterministický výběr semaforu, který bude spuštěn. Moduly tedy běží v tzv. prokládaném režimu. To znamená, že se v každém okamžiku vyhodnocují rovnice pouze jednoho (libovolného) modulu z výše uvedených modulů. Tímto způsobem jsme zajistili asynchronní souběh systémů popsaných jednotlivými moduly. Oba semafory tedy pracují paralelně, ale vzájemně zobrazují jinou barevnou signalizaci. Na jednom semaforu bude například svítit zelená a na druhém červená. Pak dojde ke změnám barev a jednotlivé barvy se vystřídají tak jak mají. Je zajištěn reálný chod dvou semaforů. Hlavní modul s klíčovým slovem *process* v definici instance vidíme na následujícím výpisu.

```
MODULE main
VAR
      semafor1 : process svetlo(semafor2.state);
      semafor2 : process svetlo(semafor1.state);
```

Použitím klíčového slova *process* jsme tedy zajistili, že oba semafory se vzájemně ovlivňují. Fungují tedy takovým způsobem, že na jednom semaforu svítí červená a na druhém semaforu dojde ke změně barevné signalizace postupně z červené na červeno-oranžovou, dále na zelenou a nakonec skrze oranžovou zpět na červenou. Na tomto semaforu pak zůstane červená a dochází ke změně na semaforu prvním. V tomto případě oba semafory měly bez použití slova *process* po dobu běhu vždy stejné barvy. Jednalo se o synchronní souběh systémů popsaných jednotlivými moduly. Asynchronní průběh lze vidět na obrázku č. 45.

Díky asynchronnímu běhu máme zajištěno, že nedojde ke kolizi v různých směrech křižovatky. Jednalo se však o velice zjednodušený model. Tento model je uveden v příloze




Obrázek 45: Barevná signalizace na semaforech.

na CD v adresáři *Příklady* pod názvem *krizovatka.smv*. Pro praktické nasazení je model v tomto stádiu nepoužitelný. Neboť kdybychom chtěli řešit křižovatku jako takovou a zabývat se všemi detaily, museli bychom zohlednit velké množství vlastností, které se reálné situace dotýkají a ovlivňují ji. Model by nabyl na velikosti a mohlo by dojít k situaci, že toto řešení a jeho případné rozšiřování není zvoleno správně a musel by se vytvořit model jiný. Pokusme se tedy podívat na část systému křižovatky a to na přechod pro chodce, který zkusíme vyřešit v následující kapitole.

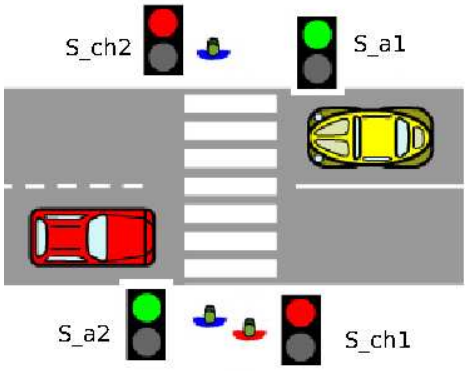
## 7.11 Přechod pro chodce

V příkladu, který bude následovat, se zaměříme na řešení modelu, který představuje přechod pro chodce. Inspiraci pro tento model jsem čerpal z online dokumentu, který popisuje řešení modelu křižovatky [36]. Následující příklad je rozdělen do částí, kdy postupně od jednodušší verze řešení postupujeme pomocí různých vylepšení a zpřesnění ke konečnému výsledku. Je zde ukázáno, že při modelování konkrétních systémů je třeba zohlednit velké množství vlastností a různých specifik systému, aby bylo možné vytvořit ke konkrétnímu systému odpovídající model.

Ukážeme si tedy řešení dopravní situace znázorněné na obrázku č. 46. Jedná se o komunikaci, přes kterou je veden přechod pro chodce. Přechod umožňuje díky dvěma semaforům bezpečný průchod chodců na opačnou stranu komunikace v době kdy je provoz na této komunikaci pro projíždějící vozidla zastaven.

Budeme tedy řešit situaci, kdy máme dva semafory pro chodce (S_ch1, S_ch2) a dva semafory pro vozidla (S_a1, S_a2). První myšlenka možného řešení, která nás může napadnout, je udělat modul pro semafor vozidel, modul pro semafor chodců a nějaký modul, který nám bude inicializovat změny stavů semaforů. Následně by bylo potřeba vytvořit dva procesy (instance) jednotlivých modulů semaforů chodců a vozidel. Získali bychom tedy dva procesy pro vozidla a dva procesy pro chodce. Dále bychom vytvořili jednu instanci (proces) modulu pro obsluhu tlačítka. Tedy zajištění změny stavů semaforů. Vytvořit model na základě popsaného není složité. Avšak při zamyšlení se nad daným problémem, případně již při řešení, nám vyvstane jedna důležitá otázka. A tou je, jak zajistit, aby při stisku tlačítka došlo k opětovné změně signalizace, tedy ke změnám stavů jednotlivých semaforů? Tuto otázku již není tak lehké vyřešit. Zkusíme si tedy nejprve celý model důkladně popsat.

Nyní si představíme odlišné řešení, než které nás napadlo dříve. Řešení, které však povede ke konečnému výsledku. Na obrázku č. 46 vidíme dva semafory pro vozidla a




Obrázek 46: Přechod pro chodce

dva semafory pro chodce. Vidíme, že všechny čtyři semafory mohou signalizovat pouze dvě barvy (červená, zelená). Jedná se o malé zjednodušení modelu, kterého následně využijeme a které nám podstatně zjednoduší práci. Dále na obrázku vidíme vozidla ve dvou směrech a chodce, kteří se chtějí z jedné strany komunikace dostat na stranu druhou. V tomto řešení nebudeme uvažovat tlačítko, které by nám inicializovalo změny. Budeme uvažovat situaci, kdy nám náhodně přijíždějí vozidla a přicházejí chodci. Nyní se podívejme na konkrétní řešení jednotlivých částí a postupně si blíže specifikujme náš model, který v závěru budeme verifikovat na konkrétní vlastnosti.

Uvedli jsme si, že v našem modelu budeme mít vozidla ve dvou směrech. Definujme si tedy dvě proměnné, které nám budou říkat, zda je vozidlo v konkrétním směru. Tyto proměnné budou pojmenovány *auto1*, *auto2*. Stejně tak i pro chodce si vytvoříme dvě proměnné. Jejich názvy budou *chodec1*, *chodec2*. Dále si vytvoříme proměnné, které nám budou představovat požadavek od vozidel a od chodců (*pozadavek_a1*, *pozadavek_a2*, *pozadavek_ch1*, *pozadavek_ch2*). Díky těmto proměnným budeme vědět, že v konkrétním směru je vozidlo, resp. chodec, který čeká na změnu signalizace, aby mohl dále pokračovat v jízdě, resp. přejít vozovku. Pro každý semafor musíme dále nadefinovat světla, která mohou svítit. To provedeme pomocí *svetlo_a1 : {zel, cer};*. Stejným způsobem nadefinujeme barevné kombinace i pro ostatní semafory. Poslední proměnné, které si nyní definujeme, budou zámky. Tyto zámky nám poslouží k uzamčení určitých stavů modulu a zabrání používání některých proměnných, které budou mít po určitou dobu konkrétní hodnotu. Zámky si pojmenujeme jako *zamek_a1a2*, *zamek_ch1ch2*. K případnému dalšímu definování proměnných přistoupíme až později. A to v době kdy jich bude zapotřebí.

Nyní si vytvoříme jednotlivé moduly a jejich části. Vytvoříme si tedy modul, který nám bude obsluhovat světelnou signalizaci na semaforech pro chodce. Modul pojmenovaný *SmerCH* vytvoříme následovně:

```
SmerCH(chodec1, chodec2, pozadavek_ch1, pozadavek_ch2,
       svetlo_ch1, svetlo_ch2, zamek_a1a2, zamek_ch1ch2)
```

Modulu *SmerCH* předáváme jako parametry informaci o tom, zda je na obou stranách přechodu nějaký chodec, dále informaci o tom, zda již v dřívější době nepožadoval chodec změnu signalizace, aby mohl přejít přes přechod pro chodce. Dalšími parametry jsou informace o aktuálním stavu světelné signalizace a o zámcích. V rámci tohoto modulu deklarujeme proměnnou pojmenovanou *stav*. Tato proměnná představující stav modulu, se může nacházet ve čtyřech různých stavech (*necinny, zacatek, cinny, konec*). Počátečním stavem této proměnné bude stav *necinny*. Stavy následující budou záviset na několika dříve uvedených proměnných. Nyní si ukážeme, jak budou vypadat následující stavy proměnné *stav* a následně si je podrobně popíšeme.

```
init(stav) := necinny;
next(stav) := case
      stav = necinny : case
             (pozadavek_ch1 | pozadavek_ch2) : zacatek;
             1 : stav;
             esac;
      stav = zacatek & (zamek_a1a2 = odemknuto) : cinny;
      stav = cinny & (!chodec1 | !chodec2) : konec;
      stav = konec : necinny;
      1 : stav;
      esac;
```

Jak již bylo řečeno počáteční stav je nastaven na hodnotu *necinny*. Následující stav se mění v případě vyvstání nějakého požadavku. Tento požadavek říká, že buď na jedné, či na druhé straně přechodu je chodec, který by chtěl přejít. Chodci mohou být na obou stranách přechodu i zároveň. Z tohoto důvodu je mezi jednotlivými požadavky zapsán logický OR. Pokud je tedy nastaven libovolný požadavek, tak dojde ke změně stavu na stav *zacatek*. Z tohoto stavu se do následujícího stavu dostaneme jen v případě, když není zamknutý blok určený vozidlům. Jedná se o vzájemné vyloučení, neboli mutual exclusion. Informace k problematice vzájemného vyloučení jsou uvedeny v online prezentaci, která se zabývá touto problematikou [37]. Snažíme se zamykat části, které nesmí nastat v systému zároveň. K tomuto účelu používáme zámků. V případě, že je blok určený vozidlům odemčen, tak přejdeme do stavu *cinny*. V opačném případě zůstáváme v aktuálním stavu. Stav *cinny* opouštíme v případě, kdy žádný chodec již nemá v úmyslu přejít přes přechod. Pokud tedy žádný chodec na začátku, resp. konci přechodu nečeká, tak přejdeme do stavu *konec*. Z tohoto stavu již v dalším kroku přejdeme do stavu *necinny* a dostáváme se do stavu, ve kterém jsme začínali.

Přechod ze stavu *necinny* do stavu *zacatek* se děje v případě nastavení požadavku. Aby byla proměnná *pozadavek_ch1*, resp. *pozadavek_ch2* nastavena, tak musí být nejprve na jedné ze stran přechodu chodec (*chodec1, chodec2*). Již dříve jsme si uvedli, že chodci budou přicházet k přechodu náhodně. K tomu, abychom zajistili náhodný přístup chodců, využijeme nedeterminismu. Následující stav proměnných *chodec1, chodec2* bude určen nedeterministicky, viz následující zápis.

```
next(chodec1) := {0, 1};
next(chodec2) := {0, 1};
```

Máme tedy zajištěno, že v následujícím stavu mohou nabývat proměnné pro chodce jedničkové, či nulové hodnoty. Chodec na přechodu je, nebo není. Po vyřešení problému s náhodným přístupem chodců k přechodu, můžeme přejít k vyřešení nastavení požadavku. Požadavek by měl být nastaven v případě, kdy je požadavek roven nule, tedy nebyl dříve nastaven a zároveň je na jedné ze stran přechodu chodec. Řešení nastavení požadavku můžeme vidět dále.

```
next(pozadavek_ch1) := case
      !pozadavek_ch1 : case
            chodec1 : 1;
            1 : pozadavek_ch1;
            esac;
      (stav = konec) : 0;
      1 : pozadavek_ch1;
      esac;
```

Pro *pozadavek_ch2* je zápis stejný. Nastavení následujícího stavu požadavků chodců jsme tedy již vyřešili. Musíme však vyřešit i resetování tohoto požadavku. To provedeme v případě, kdy je proměnná *stav* našeho modulu rovna hodnotě *konec*. Zápis vidíme ve výše uvedeném kódu. Jedná se o *(stav = konec) : 0;*. V případě, že se u přechodu objeví chodec, tak je příznak nastaven a stav modulu přejde do dalších stavů. V těchto stavech se provedou určité operace a po jejich skončení se přechází do stavu *konec*, ve kterém již není třeba, aby byl požadavek nastaven. Z tohoto důvodu se provádí jeho resetování uvedeným příkazem.

Nyní přejdeme k vyřešení zámku pro chodce (*zamek_ch1ch2*). Díky tomuto zámku uzamkneme určitou sekci, do které se nesmí přistupovat a nesmí se manipulovat s proměnnými dříve, než proběhne uzamčená sekce a následně je zámek odemknut. Zápis zámku vypadá následovně.

```
next(zamek_ch1ch2) := case
      stav = zacatek & (zamek_a1a2 = odemknuto) : zamknuto;
      stav = konec : odemknuto;
      1 : zamek_ch1ch2;
      esac;
```

Následující stav zámku je nastaven na hodnotu *zamknuto* v případě, že stav systému je roven hodnotě *zacatek* a zároveň není zamknutá část pro vozidla. Nemohou tedy být dvě části zamknuty zároveň. Odemykání se provádí v době, kdy modul je ve stavu *konec*. Proběhly tedy všechny operace, které se měly vykonat ve stavu *cinny*.

Nyní následuje ukázka řešení světelné signalizace, která je poslední částí modulu pro chodce. Uvedli jsme, že budeme v našem zjednodušeném modelu uvažovat pouze barvy zelená a červená. Blok, který nám zajistí změnu signalizace, můžeme vidět níže.

```
next(svetlo_ch1) := case
      stav = zacatek & (zamek_a1a2 = odemknuto) : zel;
      stav = konec &
             (svetlo_ch1 = cer & svetlo_ch2 = cer) : cer;
      1 : svetlo_ch1;
      esac;
```

Na semaforu je nastavená barva zelená (*zel*) když se stav modulu nachází na začátku (zacatek) a zároveň je odemknuta sekce vozidel. Můžeme tedy pomocí zámku uzamknout sekci pro chodce a následně provádět změnu signalizace. Tím dojde k vzájemnému vyloučení dvou částí, které by neměly nastat. A to těch částí, které představují svítící zelenou barvu na všech semaforech. Provedli jsme tedy definování nastavení signalizace na zelenou barvu. Musíme však vyřešit jak z této zelené barvy opětovně přejít na barvu červenou (*cer*), která byla nastavena jako výchozí hodnota. Změna signalizace ze zelené na červenou se provádí ve stavu modulu *konec*. Pro změnu signalizace musí ještě platit, že na obou semaforech již nesvítí barva zelená.

Nyní máme hotový jeden celý modul, který nám obsluhuje dva semafory na obou stranách přechodu. Dále přejdeme k tvorbě modulů obsluhující semafory vozidel. Pro každý semafor vytvoříme jeden modul se všemi částmi (zámek, světelná signalizace, vozidla, požadavky). Oba moduly budou oproti modulu pro chodce jednodušší. Budou však využívat stejných technik jako modul pro chodce. Uvedeme si zde jednotlivé části prvního modulu pro jeden směr. Druhý modul bude vypadat stejně, jen v něm budou užity proměnné týkající se druhého směru.

Vytvoříme si modul pojmenovaný *SmerA1*, kterému budeme předávat parametry: *auto1, pozadavek_a1, svetlo_a1, svetlo_a2, zamek_a1a2, zamek_ch1ch2*. Stejným způsobem jako v modulu pro chodce tak i zde budeme spoléhat na nedeterminismus a náhodně uvažovat jednotlivá vozidla před přechodem pro chodce. Nedeterministický výběr se provede skrze následující zápis.

```
next(auto1) := {0, 1};
```

Nebudeme zde již popisovat přesně, co jednotlivé příkazy znamenají. Jejich význam byl popsán na tvorbě modulu pro chodce. Zde si vypíšeme jen jejich formu a případně uvedeme rozdíly oproti dříve uvedeným konstrukcím. Máme tedy nedeterminismus pro vozidla. Pokud je vozidlo před přechodem, tak je nastaven požadavek. V následujícím zápisu je vidět nastavení požadavku a jeho případné resetování.

```
next(pozadavek_a1) := case
      !pozadavek_a1 & auto1 : 1;
      stav = konec : 0;
      1 : pozadavek_a1;
      esac;
```

Další zápis je pro konkrétní stavy modulu pro vozidla. Opět budeme využívat čtyři stavy (*necinny, zacatek, cinny, konec*). Počáteční stav je jako v předešlém případě nastaven

na *necinny*. Přechody a podmínky, za kterých je možno z jednoho stavu do druhého přejít vídíme zde:

```
init(stav) := necinny;
      next(stav) := case
             stav = necinny : case
                    pozadavek_a1 = 1 : zacatek;
                    1 : stav;
                    esac;
             stav = zacatek & (zamek_ch1ch2 = odemknuto) : cinny;
             stav = cinny & !auto1 : konec;
             stav = konec : necinny;
             1 : stav;
      esac;
```

V modulu chodců jsme si uvedli zámek pro chodce *zamek_ch1ch2*. Jeho stav (*odemknuto*) využíváme ve výše uvedeném bloku pro přechod do stavu *cinny*. Pokud jsme tedy dříve uzamkli část pro chodce, na semaforech chodců svítí zelená, tak se nyní nedostaneme do stavu *cinny* v modulu vozidel. Tímto se v bloku nastavujícím světelnou signalizaci pro vozidla *svetlo_a1* nedostaneme do situace, kdy by svítila zelená barva na semaforu pro vozidla. Máme zajištěno vzájemné vyloučení a nemůže dojít ke kolizi vozidel s chodci. V modulu vozidel nastavujeme taktéž zámek. Zámek, který nám uzamyká sekci určenou vozidlům. Řešení zamykání a odemykání je zde:

```
next(zamek_a1a2) := case
      stav = zacatek & (zamek_ch1ch2 = odemknuto) : zamknuto;
      stav = konec & !(svetlo_a2 = zel) : odemknuto;
      1 : zamek_a1a2;
      esac;
```

Posledním blokem, který budeme využívat je blok zajišťující změnu signalizace na semaforu pro vozidla. Jeho řešení je následující:

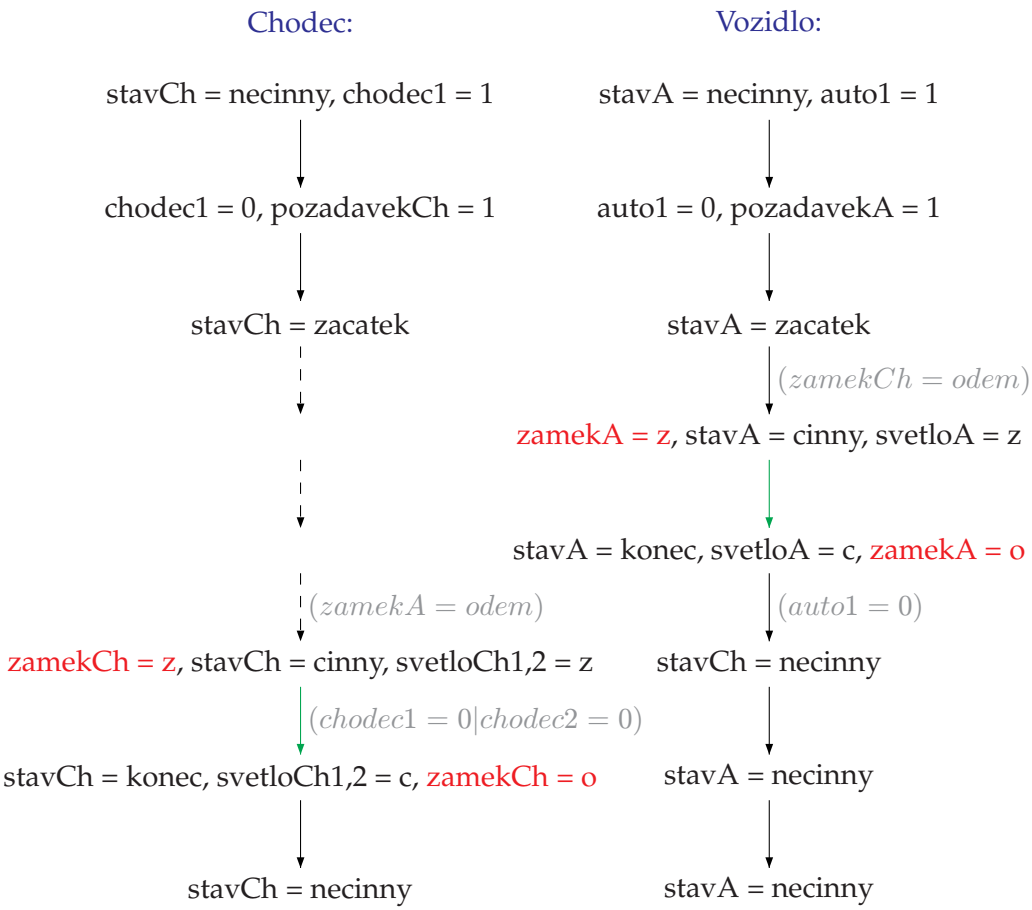
```
next(svetlo_a1) := case
      stav = zacatek & (zamek_ch1ch2 = odemknuto) : zel;
      stav = konec : cer;
      1 : svetlo_a1;
      esac;
```

Světelná signalizace z počáteční hodnoty červené (*cer*) se mění na zelenou (*zel*) jen když je stav roven *zacatek* a zároveň není uzamčená část pro chodce. Změna na původní hodnotu červenou probíhá ve stavu modulu pojmenovaném *konec*.

Tímto bychom měli vytvořený modul pro vozidla pojmenovaný *SmerA1*. Modul pro opačný směr si pojmenujeme *SmerA2* a vytvoříme jej stejným způsobem jako *SmerA1*. Jediný rozdíl bude v proměnných a to takový, že místo *auto1, pozadavek_a1, svetlo_a1,*

*svetlo_a2*, *zamek_a1a2*, *zamek_ch1ch2* použijeme *auto2*, *pozadavek_a2*, *svetlo_a2*, *svetlo_a1*, *zamek_a1a2*, *zamek_ch1ch2*. Můžeme si všimnout, že některé parametry zůstávají stejné a jen několik se změnilo.

Chování našeho modulu by mělo odpovídat ukázce na obrázku č. 47. Na tomto obrázku jsou znázorněny dva průběhy. Jeden je pro obsluhu požadavků chodců a druhý pro vozidla. Obrázek představuje stavy modelu, jednotlivé přechody a případné podmínky. Můžeme vidět, že oba průběhy se dostanou do stavu, kdy je proměnná *stav* rovna hodnotě *zacatek*. Poté následuje uzamčení jedné sekce. V době kdy se provádí uzamčená sekce, tak druhý průběh tzv. čeká. Čeká se do stavu, kdy je uzamčená část opětovně odemčená. Pak může pokračovat i sekce, která se ještě nevykonala.




Obrázek 47: Chování modelu přechodu pro chodce

Abychom ověřili chování tohoto modelu, tak musíme specifikovat vlastnosti. Mezi důležité vlastnosti můžeme uvést například to, zda nastane situace, že bude na některém ze semaforu svítit zelená. Tyto vlastnosti zapíšeme následujícím způsobem.

```
SPEC EF(svetlo_a1 = zel)
SPEC EF(svetlo_a2 = zel)
SPEC EF(svetlo_ch1 = zel)
SPEC EF(svetlo_ch2 = zel)
```

Uvedené vlastnosti říkají, zda může nastat situace, kdy svítí zelená na jednotlivých semaforech. Další vlastnosti budou ověřovat, zda je zároveň na obou semaforech pro vozidla zelená, nebo zda je na obou semaforech pro chodce zelená. Jejich zápis je zde:

```
SPEC EF(svetlo_a1 = zel & svetlo_a2 = zel)
SPEC EF(svetlo_ch1 = zel & svetlo_ch2 = zel)
```

Celý systém jsme se snažili navrhnout tak, aby nenastala situace, kdy na semaforu pro vozidla svítí zelená a zároveň na semaforu pro chodce taktéž svítí zelená. Zabránit tomuto jsme se snažili pomocí uvedených zámků. Uvedeme si tedy vlastnosti, které nám budou ověřovat, zda toto nenastane.

```
SPEC
  AG !(svetlo_a1 = zel & (svetlo_ch1 = zel | svetlo_ch2 = zel))
SPEC
  AG !(svetlo_a2 = zel & (svetlo_ch1 = zel | svetlo_ch2 = zel))
```

Na všechny vlastnosti, které byly doposud uvedeny, by měla být kladná odpověď. Náš systém tedy pracuje, tak jak jsme požadovali. Zapomněli jsme však, že jedním z požadavků na systém bylo to, že v případě vozidla, které zastavilo před přechodem pro chodce a již bylo obslouženo vozidlo ve stejném směru dříve, dojde k opětovnému obsloužení. Tato vlastnost by se dala specifikovat následovně:

```
SPEC AG(auto1 & !(svetlo_a1 = zel) -> AF(svetlo_a1 = zel))
```

Vlasnost říká, že ve všech cestách stále platí to, že pokud je v konkrétním směru vozidlo a nesvítí zelená na semaforu, pak ve všech následujících cestách platí, že světlo semaforu bude opět svítit zeleně. Při ověřování této vlastnosti nám SMV poskytne negativní odpověď. Tato vlastnost zhavaruje již ve druhém stavu. Stavu, kdy je nastavená hodnota pro vozidlo v daném směru (*auto1 = 1*). Systém se zde dostane do smyčky, ze které se nedokáže dostat a není možné pokračovat. Jak je možné tento problém vyřešit? Může nás napadnou: Vytvořili jsme tři instance (procesy). Tyto instance jsou asynchronní. U asynchronních procesů by mělo být definováno FAIRNESS omezení s proměnnou *running*. K této proměnné se přidává podmínka, která musí nastat za život každé instance nekonečněkrát. Zkusíme tedy k proměnné *running* přidat podmínku. FAIRNESS omezení pro modul *SmerA1* lze vidět zde:

```
FAIRNESS
      running & !(auto1 & (svetlo_a1 = zel))
```

Za uvedenou poměnnou *running* je podmínka, která říká, že nesmí nastat situace, kdy je ve směru vozidlo *auto1* a zároveň je barva na semaforu pro tento směr zelená (*svetlo_a1 = zel*). Pro modul zajišťující obsluhu chodců je FAIRNESS omezení následující.

```
FAIRNESS
      running &
      (!(chodec1 & (svetlo_ch1=zel)) |
      !(chodec2 & (svetlo_ch2=zel)))
```

V případě tohoto FAIRNESS omezení podmínka říká, že nesmí nastat situace, kdy je chodec (*chodec1*) v jednom směru a zároveň je signalizace na semaforu chodců zelená (*svetlo_ch1 = zel*). Toto platí i pro opačný směr. Díky FAIRNESS omezení by mělo docházet k obsloužení požadavků jak chodců, tak i vozidel ve všech směrech. Zkusíme tedy ověřit dříve uvedenou vlastnost:

```
SPEC AG(auto1 & !(svetlo_a1 = zel) -> AF(svetlo_a1 = zel))
```

Nástroj SMV opětovně vydal negativní odpověď a názorný protipříklad. Z uvedeného protipříkladu bylo vidět, že v případě kdy bylo vozidlo v jednom směru, tak systém začal pracovat. Následně byl požadavek od chodců. Byla změněna signalizace na semaforu pro chodce a celý model fungoval. Při dalším požadavku, nyní od chodce z opačného směru došlo i k obsluze tohoto požadavku. Bohužel z uvedeného protipříkladu jsme mohli vypozorovat, že požadavek, který byl na začátku dán vozidlem, nebyl nikterak obsloužen. Budeme tedy muset náš stávající systém rozšířit a přizpůsobit tak, aby i vlastnosti podobného typu byly úspěšně verifikovány. Nynější řešení je možno shlédnout v příloze na CD v adresáři *Příklady* pod názvem *prechod_a.smv*.

Při verifikování poslední uvedené vlastnosti jsme zjistili, že dochází k situaci, kdy některé požadavky vozidel, případně chodců obslouženy jsou a některé naopak obslouženy nejsou. Zkusíme tedy rozšířit náš stávající systém o proměnnou příznak. Tento příznak nám bude určovat, který požadavek se má zpracovat. Doplňme tedy náš stávající model. Nejprve začněme v modulu obsluhujícího požadavky chodců a doplníme do sekce zabývající se následujícími stavy *next(stav)* následující:

```
...
stav = zacatek & (zamek_a1a2 = odemknuto) &
((!pozadavek_a1 & !pozadavek_a2) | priznak = chodci): cinny;
...
```

Do stavu *cinny* přejdeme v případě, kdy aktuální stav je *zacatek*, dále není zamknutá sekce pro vozidla a zároveň nejsou nastaveny požadavky z obou směrů silnice určených vozidlům *pozadavek_a1, pozadavek_a2*. V případě, že není nastaven žádný požadavek vozidly, tak musí být alespoň nastaven příznak na hodnotu *chodci*. Tedy vykonává se sekce zabývající se chodci. Použili jsme tedy proměnnou příznak a dvě proměnné představující požadavky z obou směrů určených vozidlům. Tyto tři proměnné musíme doplnit k parametrům modulu *SmerCh*. Toto rozšíření, které jsme doplnili do bloku zabývajícího se následujícími stavy modulu, doplníme i do bloku, který řeší zamykání.

```
...
stav = zacatek & (zamek_a1a2 = odemknuto) &
((!pozadavek_a1 & !pozadavek_a2) | priznak = chodci): zamknuto;
stav = konec : odemknuto;
...
```

Zajistíme tak zamykání sekce chodců za lepších podmínek, než bylo v předešlém řešení. Do modulu, který právě upravujeme, ještě doplníme nastavení následujícího stavu naší nové proměnné příznak. Tato proměnná bude nabývat dvou hodnot. A to hodnoty *auta* a hodnoty *chodci*. Následující stav příznaku bude tedy v modulu chodců vypadat takto:

```
next(priznak) := case
      stav = konec & (priznak = chodci) : auta;
      1 : priznak;
      esac;
```

Příznak bude změněn na hodnotu *auta* v situaci, kdy stav modulu je roven hodnotě *konec* a zároveň je dřívější hodnota příznaku rovna *chodci*. Další stav příznaku budeme nastavovat i v modulech vozidel. Díky tomuto příznaku a jeho hodnotě (*auta, chodci*) zajistíme vzájemné vyloučení mezi vykonáváním požadavků chodců a vozidel.

Doplníme tedy oba moduly vozidel (*SmerA1, SmerA2*) o příznak. Nejprve upravíme v modulu *SmerA1* následující stav *next(stav)*. To provedeme následujícím způsobem.

```
...
stav = zacatek & (zamek_ch1ch2 = odemknuto) &
((!pozadavek_ch1 & !pozadavek_ch2) | priznak = auta): cinny;
...
```

Význam tohoto zápisu jsme si již osvětlili dříve. Není tedy podstatné znovu vše popisovat. Z tohoto důvodu přejdeme k dalším úpravám. Následuje úprava dalšího stavu zámků *next(zamek_a1a2)*. Jedná se o zámek uzamykající sekci vozidel. Jediným rozdílem oproti úpravě zámku v modulu *SmerCh* jsou požadavky vztahující se k chodcům (*pozadavek_ch1, pozadavek_ch2*) a jiná hodnota příznaku *auta*. Tuto úpravu nyní navíc použijeme i v případě změny světelné signalizace. Rozšíření můžeme vidět zde:

```
...
stav = zacatek & (zamek_ch1ch2 = odemknuto) &
((!pozadavek_ch1 & !pozadavek_ch2) | priznak = auta) : zel;
stav = konec : cer;
...
```

Může se zdát, že toto rozšíření je na tomto místě zbytečné. Jeho význam však zjistíme při posledních úpravách modelu. Zatím jej rozebírat nebudeme, neboť se ještě o tomto zmíníme. Máme tedy upraveny tyto sekce: stavy modelu, zámek a světelnou signalizaci.

Zbývá nám tedy již jen vyřešení následujícího stavu příznaku *next(priznak)*. Příznak budeme měnit na hodnotu *chodci* v situaci, kdy stav modulu je roven *konec*, příznak je do této doby nastaven na hodnotu *auta* a zároveň není požadavek od vozidla z druhého směru. Kód, který bude provádět změnu příznaku, vypadá takto:

```
next(priznak) :=
      case
            stav = konec & (priznak = auta) &
                  !pozadavek_a2 : chodci;
            1 : priznak;
      esac;
```

Závěrem ještě doplníme parametry tohoto modulu a následně stejným způsobem upravíme i druhý modul vozidel (*SmerA2*). Nyní můžeme přejít opětovně k verifikaci vlastností. Snažili jsme se tedy pomocí příznaku vyloučit situaci, kdy je vyvolán nějaký požadavek a následně není obsloužen, neboť byly vykonány jiné operace. Při verifikování modelu pomocí dříve definovaných vlastností v předešlém řešení zjistíme, že ani použití příznaku a malých úprav jsme nezajistili požadované chování. Námi vytvořené řešení je uvedeno v příloze na CD v adresáři *Příklady* pod názvem *prechod_b.smv*.

Při verifikování vlastností jsme zjistili, že při obsluze požadavku jednoho procesu dojde k obsluze jiného požadavku v době, kdy by k tomuto nemělo ještě dojít. Musíme tedy zajistit pomocí dalších úprav vykonávání jednotlivých částí procesů. Začneme nejprve u modulu vozidel, který je pojmenován *SmerA1*. Do tohoto modulu přidáme blok představující druhý příznak. Tento příznak bude nastaven v době, kdy je stav modulu roven hodnotě *konec*. Zápis této operace lze vidět v následujícím výpisu.

```
next(priznak_a1) :=
      case
            svetlo_a1 = konec : 1;
            1 : priznak_a1;
      esac;
```

Nastavení bude tedy provedeno v době, kdy již proběhla změna světelné signalizace a stav systému je roven hodnotě *konec*. Tedy požadavek vozidla na průjezd skrz přechod byl zpracován a mohou se vykonávat další požadavky. Toto rozšíření však není jediná věc, kterou provedeme v tomto modulu. Rozšíříme dříve vytvořený příznak o několik dalších podmínek. Dříve než si popíšeme jednotlivé části příznaku, tak si jej ukážeme v následujícím výpisu.

```
next(priznak) :=
      case
            stav = konec & (priznak = auta)  &
                  (!pozadavek_a2 | (priznak_a2 &
                  (pozadavek_ch1 | pozadavek_ch2))) : chodci;
            1 : priznak;
      esac;
```

V této sekci nastavujeme následující stav proměnné *priznak*. Již dříve jsme definovali, že příznak bude nastaven na hodnotu *chodci* v případě, kdy stav modulu je roven hodnotě *konec*, zároveň má příznak dříve nastavenu hodnotu *auta* a také není požadavek od vozidla z opačného směru *!pozadavek_a2*. Správně jsme určili změnu příznaku za podmínky, že již není požadavek od druhého vozidla. Tato podmínka nám však nestačí a musíme ji rozšířit. V aktuálním modulu *SmerA1* jsme před několika okamžiky vytvořili část, která řeší nastavení dalšího příznaku *priznak_a1*. Podobnou část, bude mít i modul řešící příznak pro opačný směr vozidel *SmerA2*. Nyní využijeme hodnotu příznaku (*priznak_a2*) z druhého modulu. Příznak (*priznak*) bude měnit svou hodnotu *auta* na hodnotu *chodci* jen v situaci, kdy stav modulu je roven hodnotě *konec*, příznak má dříve nastavenou hodnotu *auta* a platí, že není nastaven požadavek vozidla v opačném směru nebo je nastaven příznak *priznak_a2* a zároveň je požadavek vyvolaný chodci (*pozadavek_ch1*, *pozadavek_ch2*). Můžeme tedy ukončit vykonávání obsluhy požadavku od vozidla *A1*, neboť již není požadováno setrvání v této sekci vozidlem v opačném směru nebo nastala situace, kdy obsluha požadavku vozidla v opačném směru je již u konce a zároveň byl nastaven požadavek jedním z chodců.

Poslední částí, kterou zbývá doplnit je doplnění proměnné *priznak_a1* a *priznak_a2* do definice modulu *SmerA1*. Nyní máme rozšířen modul *SmerA1*. Stejným způsobem rozšíříme druhý modul *SmerA2*.

Závěrem musíme rozšířit i modul chodců *SmerCH*. Zde doplníme jen dva programové bloky, které nám vyresetují oba příznaky, které jsou používány v modulech vozidel. Resetování se bude provádět v době, kdy stav modulu je roven hodnotě *konec*. Rozšíření lze vidět v následujícím výpisu.

```
next(priznak_a1) :=
      case
            stav = konec : 0;
            1 : priznak_a1;
            esac;


next(priznak_a2) :=
      case
            stav = konec : 0;
            1 : priznak_a2;
      esac;
```

Mohla by nás napadnou ještě myšlenka na rozšíření podmínek pro změnu proměnné *priznak*. Provést rozšíření není špatné. Avšak zde již nemá takový význam jako v modulech vozidel. Můžeme jej po vzoru modulů vozidel rozšířit, ale také nemusíme. Záleží na vlastním uvážení. Při ověřování vlastností dojdeme ke zjištění, že všechny verifikované formule jsou pravdivé. Pro kontrolu jsou zde uvedeny vygenerované odpovědi na jednotlivé verifikované vlastnosti nástrojem SMV.

```
-- specification EF svetlo_ch1 = zel  is true
-- specification EF svetlo_a1 = zel  is true
-- specification EF svetlo_a2 = zel  is true
-- specification EF (svetlo_a1 = zel & svetlo_a2 = zel) is true
-- specification EF (svetlo_ch1 = zel & svetlo_ch2 =... is true
-- specification AG (!(svetlo_a1 = zel & (svetlo_ch1... is true
-- specification AG (!(svetlo_a2 = zel & (svetlo_ch1... is true
-- specification AG (auto1 & svetlo_a1 = cer -> AF s... is true
-- specification AG (auto2 & svetlo_a2 = cer -> AF s... is true
-- specification AG (chodec1 & svetlo_ch1 = cer -> A... is true
```

Výsledné řešení je uvedeno v příloze na CD v adresáři *Příklady* pod názvem *prechod_c.smv*.

# 8 Závěr

V této diplomové práci jsem se zabýval problematikou symbolického ověřování modelů pomocí nástrojů Symbolic model verifer (SMV) a NuSMV. Symbolické ověřování modelů je technika, která slouží pro určování správnosti a spolehlivosti programů. Tato technika využívá pro reprezentaci stavového prostoru struktury zvané binární rozhodovací diagramy. Aby bylo možné uvést různé poznatky o těchto strukturách, poznatky o symbolickém ověřování modelů a následně ukázat možnosti zmíněných nástrojů, tak bylo nejprve potřeba nastudovat problematiku zabývající se těmito oblastmi. Množství informací, které bylo potřeba čtenáři předložit je obrovské. Z tohoto důvodu byla práce rozdělena do několika kapitol. První kapitola práce byla zaměřená na základní pojmy z teorie grafů, výrokové logiky a na vysvětlení některých důležitých pojmů. Po této kapitole následuje kapitola, která uvádí informace o formální verifikaci, logikách, které se používají pro specifikaci kritérií při modelování systémů. Závěrem této kapitoly je čtenáři předložen výčet programů, které lze použít ve formální verifikaci. Nástroje SMV a NuSMV které byly používány v rámci této práce, jsou určeny pro symbolické ověřování modelů, neboli tzv. symbolický model checking. Aby bylo možné tento symbolický model checking popsat, tak bylo nejprve potřeba uvést informace o ověřování modelů jako takovém. Ověřování modelů neboli model checkingu jsem věnoval další kapitolu. V kapitole jsou uvedeny důležité poznatky o model checkingu, základních rysech této techniky a postup, jak se ověřování modelů provádí, případně jak modelovat konkrétní systém. Po kapitole zabývající se ověřováním modelů následuje kapitola zabývající se symbolickým ověřováním modelů. V této kapitole jsem podrobněji popsal struktury, na kterých jsou nástroje SMV a NuSMV založené. Jsou zde tedy uvedeny informace o binárních rozhodovacích diagramech, jejich modifikacích na uspořádané binární rozhodovací diagramy, či redukované uspořádané binární rozhodovací diagramy. V kapitole jsem taktéž uvedl popis operací, které je možné nad danými strukturami provádět. Na závěr je ještě uveden způsob reprezentace uvedených struktur pomocí knihovny CUDD. V této chvíli, kdy již byly uvedeny informace o symbolickém ověřování modelů, bylo možno uvést i několik informací o nástrojích, které jsou na symbolickém ověřování modelů založené. Těmto nástrojům, jejich podrobnějšímu popisu jsem věnoval taktéž jednu kapitolu. Po této kapitole, která dává čtenáři přehled o vybraných nástrojích, následuje poslední, avšak nejvýznamnější kapitola. Tato závěrečná kapitola této diplomové práce ukazuje na konkrétních příkladech, jak lze nástroj SMV, případně NuSMV využívat a pro které konkrétní systémy je nejvhodnější jejich použití. Pro pochopení funkčnosti a jako názorná ukázka tato tzv. příkladová studie postačuje. V rámci této práce však nebylo možné ukázat podrobně všechny možnosti nástroje SMV, NuSMV, neboť se jedná o velice komplexní problematiku. Při práci s oběma nástroji a při tvorbě modelů konkrétních systémů jsem došel k závěru, že pomocí uvedených nástrojů nelze modelovat a verifikovat časové charakteristiky systémů. Lze však provádět verifikaci konečně stavových systémů. Tedy systémů, které jsou reprezentovány pomocí různých stavů a podmínek, které řeší přechody mezi těmito stavy. V případě, že by měl uživatel zájem používat při verifikování modelů systémů časové charakteristiky těchto systémů, tak bych jej odkázal na internetové stránky zabývající se nástrojem Uppaal [38]. Tento nástroj je totiž přímo

zaměřen na řešení modelů systémů využívajících čas. Pro detailnější seznámení s nástrojem SMV doporučuji domovskou internetovou stránku tohoto nástroje [26]. Zde jsou uvedena konkrétní řešení různých problémů, vytvořených modelů reálných systémů a další materiál vhodný ke studiu.

Tato diplomová práce tedy nabízí čtenáři možnost proniknout do problematiky symbolického ověřování modelů. Předkládá mu smysluplně uspořádané informace, které je nutno znát, pro pochopení dané problematiky a zobrazuje na konkrétních příkladech jak navrhovat, vytvářet, modelovat a také verifikovat různé konečně stavové systémy. Čtenář by po přečtení této diplomové práce měl být sám schopen namodelovat a ověřit vlastní systém a mít představu o základech symbolického ověřování modelů.

Bc. Zdeněk Janoš

# 9 Literatura

[1] BRIM, Luboš, ČERNÁ, Ivana. *Správnost počítačových programů - očekávání a realita*. Brno, 2006. Multioborový seminář na fakultě Informatiky, Mysarykovy univerzity v Brně.

[2] BALCÁREK, Jiří. *Řešení problému splnitelnosti booleovské formule (SAT) pomocí binárních rozhodovacích diagramů (BDD)*. Praha, 2007. 49 s., 20 s. příloh. Bakalářská práce na Elektrotechnické fakultě ČVUT v Praze. Vedoucí bakalářské práce Ing. Petr Fišer.

[3] CHYTIL, M. *Teorie automatů a formálních jazyků*. Brno: Addison, 1982. Skriptum na PřF UJEP.

[4] ŠPRDLÍK, Otakar. *Verifikace s SMV*. Praha, 2005. 74 s., 15 s. příloh. Diplomová práce na Elektrotechnické fakultě ČVUT v Praze na katedře Řídící techniky. Vedoucí diplomové práce Ing. Richard Šusta, Ph.D.

[5] JANOŠ, Zdeněk. *Práce s verifikačním nástrojem*. Ostrava, 2007. 32 s. Bakalářská práce na fakultě Elektrotechniky a informatiky Vysoké školy Báňské - Technické univerzity na katedře informatiky. Vedoucí bakalářské práce Ing. Martin Kot.

[6] ANDERSEN, Henrik Reif. *An Introduction to Binary Decision Diagrams*. Denmark, 1999. 36 s. Lecture Notes. Department of Information Technology, Technical University of Denmark.

[7] BÉRARD, B., BIDOIT, M., FINKEL, A., LAROUSSINIE, F., PETIT, A., PETRUCCI, L., SCHNOEBELEN, Ph., MCKENZIE, P.. *System and Software Verification: Model-Checking Techniques and Tools*. Germany: Springer, 2001. 190 s. ISBN 3-540-41523-8

[8] NAVRKAL, Michal. *Detekce tautologie pomocí BDD*. Praha, 2006. 31 s., 24 s. příloh. Bakalářská práce na Elektrotechnické fakultě ČVUT v Praze. Vedoucí bakalářské práce Ing. Petr Fišer.

[9] BÍLEK, Jan. *Minimalizace neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů*. Praha, 2007. 75 s., 7 s. příloh. Diplomová práce na Elektrotechnické fakultě ČVUT v Praze. Vedoucí diplomové práce Ing. Petr Fišer.

[10] KOLOŠ, Ondřej. *Port programového balíku CUDD pod platformou Windows*. Praha, 2006. 62 s., 6 s. příloh. Bakalářská práce na Elektrotechnické fakultě ČVUT v Praze. Vedoucí bakalářské práce Ing. Petr Fišer.

[11] VÝMOLA Michal. *Porovnání softwarových nástrojů pro model checking*. Ostrava, 2008. 69 s., 3 s. příloh. Diplomová práce na fakultě Elektrotechniky a informatiky Vysoké školy Báňské - Technické univerzity na katedře informatiky. Vedoucí diplomové práce Ing. Zdeněk

Sawa, Ph.D.

[12] FELCMAN, Martin. *Porovnání dostupných programových balíků pro manipulaci s binárními rozhodovacími diagramy* Praha, 2008. 59 s. Bakalářská práce na Elektrotechnické fakultě ČVUT v Praze. Vedoucí bakalářské práce Ing. Petr Fišer.

[13] Mgr. KOVÁŘ, Petr, Ph.D. *Diskrétní matematika: Úvod do Teorie Grafů* [online]. Poslední revize 2008 [cit. 17-02-2009]. Dostupné z: <http://homel.vsb.cz/∼kov16/files/dim_prednaska06.pdf>.

[14] doc. RNDr. DUŽÍ, Marie, CSc. *Matematická logika* [online]. Poslední revize 2003 [cit. 18-02-2009]. Dostupné z: <http://www.cs.vsb.cz/duzi/Matlogika.pdf>.

[15] ALUR, R., HENZINGER, T, A. *Linear Temporal Logic* [online]. Poslední revize 07. 12. 1999 [cit. 20-02-2009]. Dostupné z: <www.cis.upenn.edu//∼alur/CIS673/12.ps>.

[16] RNDr. BARNAT, Jiří, Ph.D. *Validace a verifikace: Logiky větvícího se času* [online]. Verze 1.0. Poslední revize 25. 11. 2008 [cit. 23-02-2009]. Dostupné z: <www.fi.muni.cz/∼xbarnat/IV113/2008/IV113_11_Branching_time.pdf>.

[17] KESTEN, Yonit. *Introduction to Software Verification* [online]. [cit. 03-02-2009]. Dostupné z WWW: <http://www.cs.bgu.ac.il/∼sv041/>.

[18] SOMENZI, Fabio. *CUDD: CU Decision Diagram Package* [online]. Poslední revize 20. 02. 2009 [cit. 01-03-2009]. Dostupné z WWW: <http://vlsi.colorado.edu/∼fabio/CUDD/cuddIntro.html>.

[19] LIND-NIELSEN, J. *BuDDy: Binary Decision Diagram package* [online]. Verze 2.0. Poslední revize květen 2001 [cit. 27-02-2009]. Dostupné z WWW: <http://www.itu.dk/research/buddy>.

[20] BIERE, Armin. *The ABCD package* [online]. [cit. 27-02-2009]. Dostupné z WWW: <http://fmv.jku.at/abcd/>.

[21] RANJAN, R., SANGHAVI J. *CAL BDD package* [online]. Verze 2.1. Poslední revize 2008 [cit. 27-02-2009]. Dostupné z WWW: <http://embedded.eecs.berkeley.edu/Research/cal_bdd/>.

[22] *EDA Verification, BDDs* [online]. [cit. 13-12-2008]. Dostupné z: <http://cc.ee.ntu.edu.tw/ eda/Course/IntroEDA/LN/verification.pdf>.

[23] RNDr. BARNAT, Jiří, Ph.D. *Symbolický přístup k metodě ověřování modelu* [online]. Verze 1.0. Poslední revize 11. 12. 2008 [cit. 07-01-2009]. Dostupné z: <http://www.fi.muni.cz/∼xbarnat/IV113/2008/IV113_13_Symbolic_approach.pdf>.

[24] CRHOVÁ, J., KRČÁL, P., STREJČEK, J., ŠAFRÁNEK, D., ŠIMEČEK, P. *Databáze verifikačních nástrojů* [online]. Poslední revize 27. 1. 2009 [cit. 24-03-2009]. Dostupné z WWW: <http://anna.fi.muni.cz/yahoda/>.

[25] *School of computer science* [online]. [cit. 02-12-2008]. Dostupné z WWW: <http://www.cs.cmu.edu/>.

[26] *Model Checking @CMU - The SMV System* [online]. [cit. 02-12-2008]. Dostupné z WWW: <http://www.cs.cmu.edu/∼modelcheck/smv.html>.

[27] MCMILLAN, K. L. *The SMV System for SMV version 2.5.4* [online]. Poslední revize 06. 11. 2000 [cit. 23-11-2008]. Dostupné z: <http://www.cs.cmu.edu/∼modelcheck/smv/smvmanual.ps>.

[28] CLARKE, Edmund M. *Grand Challenge Problem: Model Check Concurrent Software* [online]. [cit. 23-02-2009]. Dostupné z: <www.cs.cmu.edu/∼emc/15817-s05/GRAND%20CHALLENGE-intro.PPT>.

[29] PANDYA, Paritosh K. *Hardware verification using model checking* [online]. Poslední revize 2002 [cit. 13-01-2009]. Dostupné z WWW: <http://www.cfdvs.iitb.ac.in/>.

[30] *Installation instructions for SMV: Instructions for Unix* [online]. Poslední revize 27. 12. 2002 [cit. 10-11-2008]. Dostupné z WWW: <http://www.itu.dk/courses/ISOT/E2003/doc/install.html>.

[31] *Installation instructions for SMV: Instructions for Windows* [online]. Poslední revize 27. 12. 2002 [cit. 10-11-2008]. Dostupné z WWW: <http://www.itu.dk/courses/ISOT/E2003/doc/install.html>.

[32] *NuSMV: a new symbolic model checker* [online]. Poslední revize 22. 05. 2007 [cit. 26-02-2009]. Dostupné z WWW: <http://nusmv.irst.itc.it/>.

[33] DRAKOS, N., HENNECKE M., MOORE R., SWAN, H. *Installation of the GNuSMV-snapshot-021002* [online]. [cit. 17-03-2009]. Dostupné z WWW: <http://nusmv.irst.itc.it/gnusmv/dload/unix/README-linux.html>.

[34] CIMETTI, A., CLARKE, E., GIUNCHIGLIA, F., ROVERI, M. *NUSMV: a new Symbolic Model Verifier* [online]. [cit. 11-04-2009]. Dostupné z: <http://www.cs.cmu.edu/∼emc/papers/Papers%20In%20Refereed%20Journals/00_symbolic_model_checker.doc>.

[35] *The NuSMV Interactive Shell Commands* [online]. [cit. 11-04-2009]. Dostupné z WWW: <http://www.cs.ualberta.ca/∼hoover/cmput660/nusmv/share/nusmv/doc/html/NuSMV_Cmd_index.html>.

[36] HIMANSHU, Jain *Traffic Light Controller: Examples in SMV* [online]. Poslední revize 2007 [cit. 24-01-2009]. Dostupné z: <www.cs.cmu.edu/∼mtschant/15414-f07/lectures/traffic.ppt>.

[37] NEWEY, Malcolm *SMV by Example: Mutual Exclusion Example* [online]. Poslední revize 30. 03. 2007 [cit. 25-01-2009]. Dostupné z: <cs.anu.edu.au/student/comp4100/lectures/SMV.4up.pdf>.

[38] *UPPAAL: home page* [online]. Poslední revize 08. 09. 2006 [cit. 01-02-2009]. Dostupné z WWW: <http://uppaal.com>.

## Obsah přiloženého CD

1. Text diplomové práce ve formátu PDF.
2. Text diplomové práce ve formátu PS.
3. Příklady modelů systémů ve formátu SMV.

# A Syntaxe a sémantika

V této části bude popsána základní syntaxe a sémantika jazyka SMV.

## A.1 Slovní konvence

Jednotlivé prvky slovní konvence jsou tvořeny základními symboly. Mezi tyto symboly patří: $A, B, C, ..., Z, a, b, c, ..., z, 0, 1, ..., 9, , -$.. Sekvence těchto symbolů musí v SMV vždy začínat abecedním symbolem. Používají se také prázdné znaky, mezi tyto znaky patří: mezera a odřádkování nového řádku. V SMV se používají taktéž komentáře, ty začínají dvěmi pomlčkami " $--$" a končí odřádkováním nového řádku.

## A.2 Výrazy

Výrazy jsou v SMV jazyku tvořeny pomocí proměnných, konstant, dále jsou tvořeny z kolekcí operátorů, mezi něž zahrnujeme i Boolean spojky, integer aritmetické operátory a case výrazy.
Syntaxe výrazů je v SMV následující:

```
expr ::
        atom                ;; symbolická konstanta
        | number            ;; číselná konstanta
        | id                ;; identifikátor proměnné
        | „!“ expr          ;; logický not
        | expr1 „&“ expr2   ;; logický and
        | expr1 „|“ expr2   ;; logický or
        | expr1 „⇒“ expr2   ;; logická implikace
        | expr1 „⇔“ expr2   ;; logická ekvivalence
        | expr1 „=“ expr2   ;; rovnost
        | expr1 „! =“ expr2 ;; nerovnost
        | expr1 „<“ expr2   ;; menší než
        | expr1 „>“ expr2   ;; větší než
        | expr1 „≤“ expr2   ;; menší nebo rovno
        | expr1 „≥“ expr2   ;; větší nebo rovno
        | expr1 „+“ expr2   ;; celočíselné sčítání
        | expr1 „−“ expr2   ;; celočíselné odčítání
        | expr1 „∗“ expr2   ;; celočíselné násobení
        | expr1 „/“ expr2   ;; celočíselné dělení
        | expr1 „mod“ expr2 ;; celočíselný zbytek
        | „next“ „(“ id „)“ ;; následující hodnota
        | set_expr          ;; set výrazy
        | case_expr         ;; case výrazy
```

Symboly id nebo identifikátor jsou symboly, které identifikují jednotlivé objekty, jako proměnné nebo definované symboly.

### A.2.1 Priorita výrazů.

Priorita jednotlivých výrazů je dána následným výpisem:

$$*, /, +, -, \text{mod}, =, !=, <, >, \leq, \geq, !, \&, |, \Rightarrow, \Leftrightarrow$$

Operátory vlevo, počínaje *, / mají nejvyšší prioritu. Nejmenší prioritu má operátor $\Leftrightarrow$. Operátory rovnosti sdružují stranu levou, mimo operátorů pro implikaci $\Rightarrow$, které sdružují stranu pravou. Kulaté závorky mohou být používané i v případě, kdy chceme zahrnout celou skupinu výrazů.

## A.3 Case výrazy - case

Case výrazy mají následující syntaxi.

case_expr ::
„case“
xpr_a1 „:“ expr_b1 „;“
...
xpr_an „:“ expr_bn „;“
„esac“

Case výraz vrací hodnotu prvního výrazu pravé strany takovou, že korespondující podmínka levé strany je pravdivá. Jestliže je tedy expr_a1 pravdivé, pak je řešení expr_b1. Dále jestliže je expr_a2 pravdivé, tak řešení je expr_b2, atd. Jestliže není žádný z výrazů na levé straně pravdivý, pak je řešení case výrazu numerická hodnota 1. V další části je uvedena praktická část zápisu case výrazu.

```
case
      barva = cer & barva_jiny_sem = cer : {cer_oran, cer};
      barva = cer_oran : zel;
      barva = zel : {zel, oran};
      barva = oran : cer;
      1 : cer;
esac;
```

V uvedené části je levá strana představována proměnnou *barva*. Tato barva reprezentuje barevnou signalizaci na semaforu. Na pravé straně máme nastaveny podmínky, které nám určují do jakého stavu se proměnná *barva* dostane. Například v případě, kdy je barva oranžová (*oran*), dochází následně ke změně stavu na barvu červenou (*cer*). Pokud žádná z pravých stran neodpovídá skutečnosti, pak bude barva červená, viz numerická hodnota 1.

## A.4 Set výrazy - set

Set výrazy mají následující syntaxi.

set_expr ::
„{“ val1 „,“ .... „,“ valn „}“
| expr1 „in“ expr2 ;; sada zahrnujici tvrzení (predikát)
| expr1 „union“ expr2 ;; sada union

Sada (set) může být definována vyjmenováním svých parametrů uvnitř složených závorek. Jednotlivé elementy sady mohou být čísla nebo symbolické konstanty. Union operátor nabízí sdružení dvou sad. Jestliže je každý argument číslem, nebo symbolickou hodnotou, která je zahrnutá v sadě, je pak nucen být unikátní sadou.

## A.5 Stavové proměnné - VAR

- Deklarace stavových proměnných je následující formy:
  **název proměnné : typ;**
- Instanciace modulů je možno provádět jako:
  - Instanciace synchronní
    **VAR**
    **název instance : název modulu(params);**
  - Instanciace asynchronní
    **VAR**
    **název instance : process název modulu(params);**

Stav modelu je přiřazení určité hodnoty sadě ze stavové proměnné. Proměnné, neboli instance modulů jsou tedy deklarovány následující notací:

decl :: „VAR“
atom1 „:“ type1 „;“
atom2 „:“ type2 „;“
...

Pro názornost uvedu konkrétní zápis sekce VAR v modulu pojmenovaném *nazev_modulu*. Je zde deklarována proměnná *stav*. Tato proměnná se může nacházet v těchto případech, typech: *S0, S1, S2, S3*. Lze tedy vidět, že je možné definovat pro konkrétní proměnnou více možných typů. Může se jednat taktéž o Boolean hodnotu, skalár, uživatelem definovaný modul, případně definované pole.

```
MODULE nazev_modulu
VAR
      stav: {S0, S1, S2, S3};
```

## A.6 ASSIGN deklarace - ASSIGN

- Soubor přiřazení tvaru:
  **stavová proměnná := výraz;**
- Inicializace proměnné x se provádí:
  **init(x) := 0;**
- Určení hodnoty proměnné x v následujícím stavu:
  **next(x) := x + 1;**
  - výraz může být množina: nedeterministické přiřazení
- Přiřazení jsou prováděna paralelně, nezáleží na pořadí přiřazovacích příkazů.
- Musí existovat alespoň jedno konsistentní uspořádání.
- Konsistentní uspořádání = referovat lze jen již dříve přiřazenou, iniciovanou proměnnou.
- V SMV nelze napsat neimplementovatelný model.

Assign deklarace má tedy následující formu:

decl :: „ASSIGN“
dest1 „:=“ expr1 „;“
dest2 „:=“ expr2 „;“
...

dest :: atom
| „init“ „(“ atom „)“
| „next“ „(“ atom „)“
...

Označení *atom* označuje aktuální ohodnocení proměnné. *init(atom)* označuje počáteční hodnotu a *next(atom)* označuje hodnotu proměnné v následujícím stavu. V další části je ukázáno, jak vypadá ASSIGN deklarace konkrétně.

```
ASSIGN
      init(barva_sem_1) := zel;
      next(barva_sem_1) := cer;

      init(barva_sem_2) := cer;
      next(barva_sem_2) := zel;
      .....
```

V tomto případě jsme inicializovali barvu prvního semaforu na zelenou a barvu druhého semaforu na červenou. V následujícím stavu došlo ke změně barev. V případě následujícího stavu je možné použít i case výraz, jak můžeme vídět dále. Počáteční stav bude nastaven na barvu červenou a stav následující bude nastaven na konkrétní barvu podle splnění daných podmínek.

```
ASSIGN
     init(barva) := cer;
     next(barva) := case
           barva = cer & dalsi_barva = cer : {cer_oran, cer};
           barva = cero_ran : zel;
           .....
     esac;
```

## A.7 DEFINE deklarace

Slouží k definování například bližšího popisu určitého stavu. Přiřazuje určitému výrazu konkrétní symbol.

- Jedná se o soubor definic signálů tvaru:
  **název signálu := výraz;**
- Signály v SMV jsou nestavové proměnné.
- Signály v SMV nejsou staticky typované.
- Sémantický check implementovatelnosti je podobný jako v ASSIGN.

DEFINE deklarace má tedy následující formu:

decl :: „DEFINE“
atom1 „:=“ expr1 „;“
atom2 „:=“ expr2 „;“
...

V příkladu může vypadat definice například takto:

```
DEFINE
      x := (state = s2 | state = s5 | state = s9);
      .....
```

Zde jednotlivým stavům s2, s5 a s9 definujeme symbol *x*. Tohoto je možno využít například při ověřování vlastnosti ve specifikaci SPEC, která ověřuje, zda se může systém nacházet ve stavu označeném *x*.

## A.8 SPEC deklarace

Sekce SPEC umožňuje zadání temporálních podmínek. Každá temporální podmínka musí splňovat všechna chování systému, tj. chování ze všech iniciálních stavů. Nástroj SMV ověřuje, zda daná podmínka je, či není splněna. V případě, že by námi zadaná podmínka splněna nebyla, tak vrátí protipříklad. Protipříklad bude navrácen jen v případě, že jej lze vytvořit. Pokud bychom v sekci SPEC napsali více deklarací, tak je jejich zápis brán konjunktivně. V případě, že bychom zapisovali vlastnosti, které jsou spjaté se stavovými proměnnými některého z modulu, tak se uvádějí v rámci definice toho zmíněného modulu. V modulu *main* se zapisují vlastnosti globálního typu. Syntaxe specifikace je následující:

decl :: „SPEC“ ctlform

CTL formule může mít tuto syntaxi:

ctlform ::

| | |
|---|---|
| expr | ;; Boolean výraz |
| \| „!“ ctlform | ;; logická negace |
| \| ctlform1 „&“ ctlform2 | ;; logický and |
| \| ctlform1 „\|“ ctlform2 | ;; logický or |
| \| ctlform1 „⇒“ ctlform2 | ;; logická implikace |
| \| ctlform1 „⇔“ ctlform2 | ;; logická ekvivalence |
| \| „E“ pathform | ;; existenční kvantifikátor |
| \| „A“ pathform | ;; všeobecný kvantifikátor |

Nyní nám zbývá již jen definovat syntaxi formule cesty, neboli pathform:

pathform ::

| | |
|---|---|
| „X“ ctlform | ;; v dalším stavu |
| „F“ ctlform | ;; eventuálně |
| „G“ ctlform | ;; globálně |
| ctlform1 „U“ ctlform2 | ;; dokud |

Konkrétní zápis specifikace může vypadat následovně.

```
SPEC AG!(semafor1.barva = zel & semafor2.barva = zel)
```